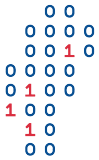


Promotionszentrum
Angewandte Informatik
HAW Hessen

Self-Supervised Learning on Source Code to Assist Software Developers

Dem Promotionszentrum
Angewandte Informatik des Landes Hessen
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
eingereichte Dissertation
von

Johannes Villmow, M.Sc.

Betreuer

Prof. Dr. Ulrich Schwanecke	Hochschule RheinMain
Prof. Dr. Adrian Ulges	Hochschule RheinMain

Gutachter

Prof. Dr. Bernhard Humm	Hochschule Darmstadt
Prof. Dr. Damian Borth	University of St.Gallen, Switzerland

Partnerhochschulen

Hochschule RheinMain
Hochschule Darmstadt
Hochschule Fulda
Frankfurt University of Applied Sciences

Einreichungstermin: 23.10.2024
Prüfungstermin: 02.04.2025

Wiesbaden 2025

Johannes Villmow October 23, 2024

Self-Supervised Learning on Source Code to Assist Software Developers

Abstract

This dissertation addresses two challenges inherent in software development: code reuse and quality. It explores the use of Machine Learning (ML), specifically transformer models, to address these challenges. The transformer architecture has recently become the de facto standard in Natural Language Processing (NLP). While code could be treated similar to natural language text, it offers unique opportunities and challenges: it follows a rigid syntax, combines structural with natural language elements, has operational semantics (i.e., code executes), and hence requires a different "understanding" than natural language. To this end, this thesis develops strategies to incorporate the structural properties of source code into transformer models, and improves tasks that require semantic code understanding of ML models, such as code summarization, code retrieval, and quality assessment of code identifiers. The first task, *code summarization*, generates natural language descriptions for code snippets, which assists developers at the tedious task of writing documentation. The second task investigates a strategy for *code retrieval*, called Contextualized Code Search (CCS), that aims at an opportunistic code reuse, by allowing the developer to retrieve relevant code snippets from a codebase based on a developer's current coding context and cursor position alone. The third task explores assessing the *quality of identifiers* in source code based on established coding guidelines and ML models, which are essential for code readability and maintainability.

The dissertation is organized into two parts: Part I *Models and Techniques*, and Part II *Applications*. The first part introduces novel approaches to integrate structural information from Abstract Syntax Trees (ASTs) into transformer models. Chapter 3 extends relative positional embeddings to encode structural relationships between nodes in the AST and proposes a new structure-aware loss function based on predicting the Lowest Common Ancestor (LCA) of nodes in trees. Chapter 4 follows a different strategy to integrate structural information, namely through a structure-aware pretraining. It presents a novel self-supervised structural pretraining task called tree-based span selection, which selects spans for masking based on the AST, and suggests improvements to the structural task identifier deobfuscation. The main result is a multi-task encoder-decoder Language Model (LM), called SyntaxPT, that achieved state-of-the-art code understanding performance

at the time of development. Finally, Chapter 5 studies the application of SyntaxPT to code retrieval, specifically Contextualized Code Search (CCS). The chapter develops a self-supervised training strategy and model for CCS, SyntaxPT-ccs, and introduces the Cocos (COntextualized COde Search Dataset) dataset, the first dataset of its kind allowing to directly evaluate CCS performance. The chapter demonstrates that SyntaxPT-ccs outperforms traditional keyword-based retrieval methods.

The second part of this thesis applies the SyntaxPT and SyntaxPT-ccs models to two highly relevant tasks in software development, code reuse and identifier quality assessment, and investigates the aforementioned models' utility from a practical perspective. Chapter 6 introduces CodeBuddy, a prototype application for CCS, and introduces enhancements to the pretraining strategy to improve retrieval robustness when interacting with end-users. The chapter validates the effectiveness through two user studies: a controlled experiment with 41 computer science students and a three-month case study with four professional software developers. The final Chapter 7 explores the application of the SyntaxPT model to assess the quality of identifiers in source code. To do so, it introduces novel self-supervised scoring functions based on the likelihood estimated by the LM to detect violations of established identifier naming guidelines. The chapter introduces the first dataset for assessing identifier quality based on coding guidelines and demonstrates that the SyntaxPT model outperforms other state-of-the-art language models on this task.

Johannes Villmow 23. Oktober 2024

Self-Supervised Learning on Source Code to Assist Software Developers

Zusammenfassung

Diese Dissertation befasst sich mit zwei wesentlichen Bestandteilen der Softwareentwicklung: Code-Reuse und Code-Qualität. Dafür untersucht diese Arbeit den Einsatz von maschinellem Lernen (ML), insbesondere von Transformermodellen. Die Transformer-Architektur ist in letzter Zeit zum de facto Standard im Bereich der Verarbeitung natürlicher Sprache (NLP) geworden. Obwohl Code ähnlich wie natürlichsprachlicher Text behandelt werden könnte, bietet er doch einzigartige Möglichkeiten und Herausforderungen. Er besteht aus einer klar definierten Syntax, kombiniert syntaktische mit natürlichsprachlichen Elementen, hat eine operative Semantik (d.h. Code kann ausgeführt werden) und erfordert daher eine andere Art von "Verständnis" als natürliche Sprache. Daher werden in dieser Arbeit Strategien entwickelt, um die syntaktischen Eigenschaften von Programm-Code in Transformermodellen nutzbar zu machen. Dabei wird versucht die Performance in Anwendungsgebieten zu verbessern, die ein semantisches Codeverständnis von ML-Modellen erfordern, wie beispielsweise automatische Dokumentierung von Code, Code-Suche und Qualitätsschätzung von Bezeichnern im Code. Im ersten Anwendungsgebiet, der Erzeugung von Dokumentationen, werden natürlichsprachliche Beschreibungen für Code-Passagen generiert. Im zweiten Anwendungsgebiet wird eine neuartige Strategie für die Code-Suche untersucht: die kontextualisierte Codesuche (CCS). CCS findet allein basierend auf dem Code im Editor und einer Cursorposition hilfreiche Code-Passagen in einer Codebasis. So ermöglicht CCS eine opportunistische Wiederverwendung von Code und einen niederschwelligen Austausch zwischen Entwicklern. In einem dritten Anwendungsgebiet wird die Nutzung von ML-Modellen zur Schätzung der Qualität von Bezeichnern im Code untersucht, basierend auf etablierten Guidelines. Bezeichner sind ein wesentlicher Faktor für die Lesbarkeit und Wartbarkeit des Codes und daher von großer Bedeutung für die Softwarequalität.

Die Dissertation gliedert sich in zwei Hauptteile: Teil I *Modelle und Techniken*, und Teil II *Anwendungen*. Der erste Teil untersucht neuartige Ansätze, um die syntaktischen Informationen aus Syntaxbäumen (ASTs) in Transformermodellen nutzbar zu machen. Dafür erweitert Kapitel 3 die Transformerarchitektur um relative Positionsembeddings, die es ermöglichen die strukturellen Beziehungen zwischen Knoten im AST zu kodieren und schlägt eine neue Lossfunktion vor, die den kleinsten gemeinsamen Vorfahren (Lowest

Common Ancestor - LCA) zweier Knoten vorhersagt. Kapitel 4 verfolgt durch ein syntaxbasiertes Pretraining eine andere Strategie, um die syntaktischen Informationen nutzen zu können. Es wird ein neuer selbstüberwachter struktureller Pretrainingtask vorgestellt, bei dem Teilelemente basierend auf dem Syntaxbaum maskiert werden. Zudem wird ein weiterer struktureller Pretrainingtask verbessert, bei dem Bezeichner maskiert werden. Der Hauptbeitrag ist ein Encoder-Decoder-Sprachmodell (LM), genannt SyntaxPT, das mit verschiedenen strukturellen und regulären Pretrainingtasks trainiert wird und zum Zeitpunkt der Entwicklung herausragende Ergebnisse auf Codeverständnis Benchmarks erzielte. In Kapitel 5 wird schließlich die Anwendung von SyntaxPT für Code-Suche untersucht, insbesondere für kontextualisierte Codesuche (CCS). Hauptbeitrag des Kapitels ist eine selbstüberwachte Pretrainingsstrategie und ein Modell für CCS, genannt SyntaxPT-ccs. Außerdem wird ein Benchmark-Datensatz namens Cocos vorgestellt. Cocos ist der erste verfügbare Datensatz für CCS, der eine direkte Evaluation von CCS-Modellen ermöglicht. Es wird gezeigt, dass das selbstüberwachte SyntaxPT-ccs Modell traditionelle schlagwortbasierte Suchmethoden übertrifft.

Der zweite Teil dieser Arbeit wendet die Modelle SyntaxPT und SyntaxPT-ccs auf den zuvor genannten Themengebieten in der Softwareentwicklung an und untersucht den Nutzen der genannten Modelle aus praktischer Sicht: (1) Wiederverwendung von Code mit CCS und (2) die Bewertung der Qualität von Bezeichnern. Dafür wird in Kapitel 6 CodeBuddy vorgestellt, ein Prototyp für die Interaktion mit dem CCS-Modell. Außerdem werden Verbesserungen der Pretraining-Strategie vorgestellt, um die Robustheit des CCS-Modells bei der Interaktion mit echten Nutzern zu verbessern. Des Weiteren wird in diesem Kapitel die Nützlichkeit anhand zweier Nutzerstudien evaluiert. Dies erfolgt zum einen anhand eines kontrollierten Experiments mit 41 Informatikstudenten und zum anderen anhand einer dreimonatigen Fallstudie mit vier professionellen Softwareentwicklern. Das abschließende Kapitel 7 untersucht die Anwendung des SyntaxPT-Modells für die Schätzung der Qualität von Bezeichnern in Programm-Code. Dafür werden neuartige selbstüberwachte Bewertungsfunktionen vorgestellt, die die Wahrscheinlichkeiten des Sprachmodells nutzen, um Verstöße gegen etablierte Guidelines zur Benamung von Bezeichnern zu erkennen. Das Kapitel stellt den ersten Datensatz zur Bewertung der Qualität von Bezeichnern auf der Grundlage von Guidelines vor und zeigt, dass das SyntaxPT-Modell andere moderne Sprachmodelle bei dieser Aufgabe übertrifft.

Contents

Für dich Ole.

Danksagung

Zuallererst möchte ich mich bei meinen Betreuern Prof. Dr. Ulrich Schwanecke und Prof. Dr. Adrian Ulges, bedanken. Eure fachliche Betreuung, das wertvolle und konstruktive Feedback und die Anmerkungen zu meiner Dissertation sowie den zugehörigen Publikationen haben maßgeblich zur Qualität dieser Arbeit beigetragen. Mein größter Dank gilt jedoch dir Adrian. Du hast mich gemeinsam mit Prof. Dr. Dirk Krechel erstmals für die Forschung begeistert. Zunächst hast du mir nicht nur ein relevantes Thema für meine Masterarbeit gegeben, sondern mir auch die Möglichkeit eröffnet, das Ergebnis auf der AAAI zu präsentieren (Shah* and Villmow* et. al. 2019). Das hat mein Interesse für Machine Learning und Deep Learning nachhaltig geweckt. Gerne denke ich an den gemeinsamen Aufenthalt 2017/2018 am DFKI in Kaiserslautern während meiner Masterarbeit zurück. Außerdem möchte ich besonders für deine Unterstützung in allen Phasen meiner Promotion danken. Deine Geduld, dein großes Verständnis und die unzähligen, hilfreichen Diskussionen haben mich stets motiviert und vorangebracht. Deine Anmerkungen haben meinen wissenschaftlichen Schreibstil geprägt und mir beigebracht, worauf es in der Wissenschaft ankommt.

Des Weiteren bedanke ich mich bei meinen Gutachtern Prof. Dr. Bernhard Humm und Prof. Dr. Damian Borth für ihre Bereitschaft, Zeit und Mühe, die sie in die Bewertung meiner Arbeit investiert haben, was nicht selbstverständlich ist.

Ein großer Dank geht an meine Kollegen der Arbeitsgruppe LAVIS: Felix Hamann, Marco Wrzalik, Viola Campos, Maurice Falk, Markus Eberts, Marcel Lamott, Melina Meyer, Micha Selak und Julian Eversheim. Der wissenschaftliche Austausch mit euch sowie die gemeinsame Zeit, ob in der Arbeit oder privat, waren für mich sehr bereichernd. Die persönlichen Freundschaften, die angenehme Arbeitsatmosphäre und die gemeinsamen Konferenzen, Workshops, Spieleabende, Sportaktivitäten und Ausflüge werde ich in bester Erinnerung behalten.

Ein weiterer Dank geht an meine Freunde und Familie, die mich während dieser Zeit unterstützt haben, allen voran an meine Frau. Deine Geduld und dein Verständnis wurden oft auf die Probe gestellt – insbesondere während der intensiven Arbeitsphasen und der vielen arbeitsreichen Wochenenden. Deine Motivation hat mich auch in schwierigen Zeiten angetrieben und ich schätze es sehr, dass du mir in dieser Phase beigestanden hast, obwohl unser Sohn gerade geboren war. Zuletzt möchte ich meinen Eltern von Herzen danken. Ohne eure Unterstützung und euer Vertrauen in mich wäre diese Dissertation nie geschrieben worden. Ihr haben mir nicht nur die Möglichkeit gegeben zu studieren und zu promovieren, sondern mich auch stets ermutigt und an mich glauben lassen.

Notations

Abbreviations, terms, and metrics, such as AI, dropout, and BLEU, are highlighted in dark red color and hyperlinked to their respective entry in the list of abbreviations in Sections 9.1 to 9.5. Additionally, all models and datasets that are referred to more often—such as Bert and CodeXGlue—are also highlighted, linked, and listed with the corresponding reference in the glossary at the end of this thesis.

This thesis employs the mathematical notations and symbols listed below. Indexing variables, such as i, j, and n, and other frequently used variables are only valid in the scope of the respective section, or until they are redefined. Other notations and symbols are defined in the text where they first appear.

This list of mathematical notations is adapted from:

Ian J. Goodfellow et al. (2016). Deep Learning. Adaptive computation and machine learning. MIT Press. ISBN: 978-0-262-03561-3

Scalars, Vectors, Matrices, and Tensors

a	A scalar (integer or real) if not specified otherwise.
$\boldsymbol{a}$	A vector.
$\boldsymbol{A}$	A matrix.
$\mathbb{1}_n$	A vector filled with ones of length n.
$\mathbb{1}_{nn}$	A matrix of ones of size $n \times n$.

Sequences and Datasets

$\boldsymbol{x} = (x^{(1)}, x^{(2)}, \ldots, x^{(n)})$	A sequence of n scalars.
$\boldsymbol{x} = (\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)})$	A sequence of n vectors.
$x^{(i)}$ or $\boldsymbol{x}^{(i)}$	The i-th example from a sequence.
$\boldsymbol{x}^{(<i)}$	The subsequence of elements before the i-th element, excluding $x^{(i)}$.
y or $\boldsymbol{y}$	The target associated with $\boldsymbol{x}$ for supervised learning.
$\hat{y}^{(i)}$ or $\hat{\boldsymbol{y}}^{(i)}$	The prediction associated with $\boldsymbol{y}^{(i)}$.

Sets and Set Notation

$\mathbb{A}$	A set.
$\{0, 1, \ldots, n\}$	The set of all integers between 0 and n.
$\lvert\mathbb{A}\rvert$	The cardinality of set $\mathbb{A}$.
$\mathbb{R}$	The set of real numbers.
$\mathbb{N}$	The set of natural numbers.
$\mathbb{V}$	The vocabulary of a model (all tokens the model can predict).

Indexing and Elements

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1.
a_{ij}	A scalar associated with the i-th and j-th examples defined by the context.
$\boldsymbol{a}_{ij}$	A vector associated with the i-th and j-th examples defined by the context.
$A_{i,j}$	Element (i, j) of matrix $\boldsymbol{A}$.
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$, sometimes denoted as $\boldsymbol{A}_i$.
$\boldsymbol{A}_{:,i,j}$	1D slice of a 3D tensor at index i and j (vector).

Vector and Matrix Operations

$\boldsymbol{A}^\top$	Transpose of $\boldsymbol{A}$.
$\boldsymbol{a}^\top\boldsymbol{b}$ or $\boldsymbol{a}^\top \cdot \boldsymbol{b}$	Dot product of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$.
$\boldsymbol{A}\boldsymbol{B}$ or $\boldsymbol{A} \cdot \boldsymbol{B}$	Matrix product of $\boldsymbol{A}$ and $\boldsymbol{B}$.
$\lVert\boldsymbol{a}\rVert$	Euclidean norm of vector $\boldsymbol{a}$.
$\boldsymbol{a} \oplus \boldsymbol{b}$	Concatenation of vectors/sequences $\boldsymbol{a}$ and $\boldsymbol{b}$.
length$(\boldsymbol{a})$	Number of elements in vector $\boldsymbol{a}$.

Intervals and Numbers

$[a, b]$	The real interval including a and b.
$(a, b]$	The real interval excluding a but including b.
k	Thousand, 10^3. For example, $1k = 1{,}000$.
M	Million, 10^6. For example, $1M = 1{,}000{,}000$.
B	Billion, 10^9. For example, $1B = 1{,}000{,}000{,}000$.

Probability and Statistics

$P(\mathrm{a})$ or $p(\mathrm{a})$	Probability distribution over a discrete or random variable.
$\mathrm{a} \sim P$	Random variable a has distribution P.
$\mathbb{E}_{\mathrm{x}\sim P}[f(x)]$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$.
$D_{\mathrm{KL}}(P \parallel Q)$	Kullback-Leibler divergence between P and Q.
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$.
Pois(λ)	Poisson distribution with rate λ, i.e., $\frac{\lambda^k e^{-\lambda}}{k!}$.

Functions and Operators

$f : \mathbb{A} \mapsto \mathbb{B}$	Function f with domain $\mathbb{A}$ and codomain $\mathbb{B}$.
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (for brevity the argument $\boldsymbol{\theta}$ is often omitted.)
$\equiv$	Equivalent to.
$\log x$	Natural logarithm of x.
$\exp x$	Exponential function of x.
$\min(x, y)$	Minimum of x and y.
$\max(x, y)$	Maximum of x and y.
$\text{abs}(x)$	Absolute value of x.
$\text{ReLU}(x)$	Rectified Linear Unit, $\max(0, x)$.
$\sigma(x)$	Logistic sigmoid, $\frac{1}{1+\exp(-x)}$.
$\text{softmax}(\boldsymbol{x})$	Softmax function, for the i-th dim $\frac{\exp(x_i)}{\sum_j \exp(x_j)}$.
$\text{dropout}(\boldsymbol{x})$	Dropout function, where each element of $\boldsymbol{x}$ is zeroed out with probability p.
$\mathbf{1}_{\text{condition}}$	Equals 1 if the condition is true, 0 otherwise.
$\text{round}(a)$	Rounds a to the nearest integer.
$\lceil a \rceil$	Ceiling function; smallest integer greater than or equal to a.
$\lfloor a \rfloor$	Floor function; largest integer less than or equal to a.
$\arg\max_x f(x)$	Argument that maximizes $f(x)$.
$\arg\text{top5}_x f(x)$	Top 5 arguments that maximize $f(x)$.

Sometimes a function f accepting a scalar as an argument is applied to a vector or matrix: $f(\boldsymbol{x})$ or $f(\boldsymbol{X})$. This denotes an element-wise application of f to the array.

Good programmers know what to write. Great ones know what to rewrite (and reuse).

— Raymond 1998, p. 24

1 Introduction and Motivation

Artificial Intelligence (AI) is increasingly transforming knowledge-based work across all types of industries, for example IT and engineering, the educational sector, logistics, and health care. One prominent example of such knowledge-intensive activities is software development. Developers need to understand and manipulate large codebases, reuse code, transfer experience knowledge from one project to another, hunt bugs, come up with good architectures, structure software to avoid pitfalls, and finally, the tedious task of documenting software components. With the ever-growing reliance on software in all aspects of society, the demand for skilled programmers continues to rise. However, while the demand for software developers is increasing, the supply of skilled programmers is not keeping pace. According to recent industry reports, this shortage is expected to grow (Bitkom 2024). Bitkom (2024) estimates that the number of unfilled positions in the IT sector in Germany alone will rise from 150,000 in 2024 to 660,000 by 2040. Therefore, tools and methodologies that can assist developers and enhance their productivity and efficiency are of high societal and economic importance.

Software development is inherently complex and time-consuming, and software projects have a chance of over 60% of failure or significant additional costs (The Standish Group 2013). A considerable portion of a developer's time is spent not only on writing new code but on reading, understanding, and reusing existing code. The time pressure and resource shortages in customer-related projects leave only little time for detailed documentation, which makes developing software components that address recurring software issues in a sustainable and cross-project manner very challenging. Often enough, changes are only implemented in the code and the existing documentations are not updated. A lot of information is therefore only available in the code itself. Additionally, in modern agile

software development, code is implemented under strict deadlines and resource constraints mostly within small, independent teams (The Standish Group 2013). This often leads to the situation that good existing solutions are forgotten and unnecessarily redeveloped—with the same errors and associated costs.

Therefore, an important issue in software development is the *reuse* of existing components and solutions. Frakes and Nejmeh (1987) found that software reuse can not only effectively assist developers, increase their productivity, but also enhance the software quality. By increasing reliability and maintainability, reuse can also reduce the time and cost of software development (Grechanik et al. 2007). However, identifying and integrating reusable code components remain challenging tasks due to the sheer volume of available code and the nuances of different programming contexts. The central source of knowledge for reuse are the software company's code repositories, i.e., the codebase or collection of source code, its development over time in the form of commits, associated issues, and requirements in the form of user stories. In order to reuse, software architects and developers (1) manually search for suitable program parts, and (2) revise these and adapt them to the current issue. Both steps are often time-consuming.

Another important aspect of software development is the *quality of the code*. High-quality code is essential for the maintainability and sustainability of software projects. One aspect of code quality is the choice of identifiers: identifiers are the part of source code that developers can directly influence and interact with. They are typically written in natural language, are used to embed domain concepts, and have been found to be essential for communication between developers and code readability (Fakhoury et al. 2020). Good identifiers can significantly improve code readability and maintainability, while poor identifiers can lead to confusion and errors. Many studies have demonstrated that low-quality identifiers negatively impact code comprehension time (Deissenboeck and Pizka 2006; Fakhoury et al. 2020), which can directly increase development costs. To this end, tools that can automatically assess the quality of identifiers in source code can assist developers in writing better code and improve the overall quality of software projects.

Machine Learning (ML), and more specifically deep learning, has emerged as a powerful tool to address these challenges in software development. With ML techniques, models can build that assist developers in generating code, identifying reusable components in large codebases, and detecting quality issues. This field of research is known as machine learning for software engineering (Amershi et al. 2019), and has gained significant attention in recent years. It is closely related to the field of Natural Language Processing (NLP), that has also seen significant advancements, particularly with the development of the transformer architecture (Vaswani et al. 2017) —which has been developed in 2017, short before the work on this thesis started in 2018. This is mainly due to transformers' ability to capture long-range dependencies in sequences, to parallelize with respect to

sequence length, and to store large amounts of knowledge/statistical patterns in its weights. The success of transformers has been boosted through research that found pretraining extremely effective (Radford et al. 2018; Devlin et al. 2019). Over the course of this thesis, pretraining has shown to be one of the most important techniques in NLP and code modeling. Pretraining on large unlabeled datasets allows models to learn general representations that can be fine-tuned for specific tasks with limited labeled data. Following this development in NLP, pretrained transformers have also been applied to code understanding: CodeBert (Feng et al. 2020) and CodeT5 are two prominent examples. The most recent models understand both code and text: when research showed that pretraining of even larger model sizes, with even more training data continues to improve performance, NLP researchers acquired more diverse datasets that contain not only text but also large amounts of code (Gao et al. 2021a), culminating in Large Language Models (LLMs), such as CodeX (Chen et al. 2021), GitHub Copilot (GitHub 2024), or ChatGPT (OpenAI 2024a), showing good understanding for source code. Although, the capabilities of such LLMs are impressive, they are not error proof and have been found to often hallucinate, especially when synthesizing text from their weights (Huang et al. 2023). Integrating factual knowledge in the prompt using Retrieval-Augmented Generation (RAG) has been found to reduce their hallucinations (Ram et al. 2023).

1.1 Contributions and Outline

This thesis is located in the research field machine learning for software engineering, and as outlined in the previous section, it explores the above key-findings—the transformer architecture and large-scale pretraining—which it adapts to the domain of source code. Its research has focused on two aspects:

- Part I of this thesis aims to develop novel strategies that can effectively incorporate the structural properties of source code into Machine Learning (ML) models. This is an aspect of source code that the aforementioned Large Language Models (LLMs) and others, that treat code as token sequences, neglect. Source code is—contrary to natural language text—highly repetitive, structural, and operational (Hindle et al. 2012), which makes it fundamentally different from natural language. This stems from the code's syntax and semantics that are defined by formal grammars (much more rigorously than for natural language), which allow it to be represented as Abstract Syntax Trees (ASTs).
- Part II applies the strategies apart from research benchmarks that provide only a narrow view of the model's practical utility to two practical usage scenarios in software development: code retrieval and identifier quality assessment.

Figure 1.1 provides an overview of the thesis structure and the models developed in this thesis. The two gray blocks represent the two primary parts of the thesis: Part I "*Models*

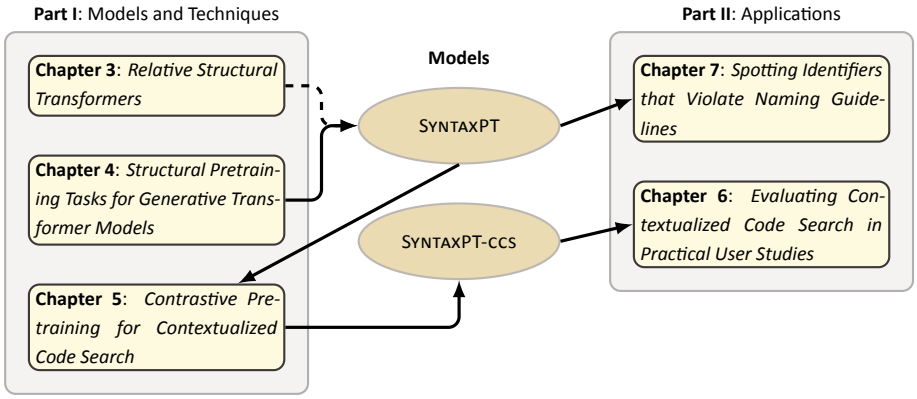


Figure 1.1: The thesis is divided into two parts: **Part I: Models and Techniques** (left) and **Part II: Applications** (right). Part I focuses on developing novel models and techniques to incorporate structural information from source code into transformer architectures, resulting in two novel ML models: SyntaxPT (which is an encoder-decoder model) and SyntaxPT-ccs (which is an encoder model for code retrieval). Part II applies the developed models to real-world problems in software development. The arrows indicate which model is presented in which chapter and how they are used in the applications.

and Techniques" and Part II "*Applications*". The main goal of the first part (Chapters 3 to 5) is to improve the semantic understanding of source code using transformer-based models by integrating structural information inherent in programming languages. This results in two major models (ellipses in Figure 1.1): A generative encoder-decoder transformer model for code, called SyntaxPT (Chapter 4), and an encoder-based transformer model for code retrieval called SyntaxPT-ccs (Chapter 5). The second part develops the CodeBuddy application that applies the SyntaxPT-ccs model to assist developers in finding relevant code snippets in their company's codebase (Chapter 6). Finally, the SyntaxPT model is applied to assess the quality of identifiers in source code (Chapter 7).

The thesis starts with an overview of the fundamental concepts in Chapter 2 and concludes with a summary of the contributions, limitations, and future work in Chapter 8. In the following sections the individual contributions to research of each chapter are presented in detail.

1.1.1 Part I: Models and Techniques

Part I focuses on developing novel models and techniques to incorporate structural information from source code into transformer models and aims to improve their performance on code understanding benchmarks.

Chapter 3: Relative Structural Transformers

In Chapter 3, we address the challenge of adding a structural prior to transformer models, which are inherently designed for sequential data. At the time of work on this chapter,

transformers had only been used as baselines for specialized code models, and their performance lacked behind state-of-the-art models (Alon et al. 2019a). This chapter introduces a novel approach that extends relative positional embeddings to encode structural relationships between nodes in ASTs. Specifically, it explores two relative positional patterns: the path length between nodes and the explicit encoding of upwards and downwards steps in the tree. Additionally, the chapter proposes a novel structure-aware loss function based on predicting the Lowest Common Ancestor (LCA) of nodes, which encourages the model to preserve structural information in its hidden states. This loss can be added to any type of ML training that encodes trees, and is not limited to transformers. The key contributions of this chapter are:

- Introduce a new relative positional tree pattern that represent movements within the tree structure and derive an efficient implementation for it.
- Propose a novel loss function that predicts the LCA of nodes in trees.
- Demonstrate that various code-to-sequence tasks can be effectively approached end-to-end[1] using self-attention-based transformers on trees with relative positional embeddings and the aforementioned contributions.
- Achieve a 6% improvement over the state-of-the-art on commonly used code understanding tasks, such as method naming and code summarization.

This work results in a model called Relative Structural Transformer (Rst), which processes tree-structured data efficiently and effectively, and addresses a direct limitation of the transformer architecture in processing hierarchical data. Integrating structural priors can reduce the amount of training data required when training end-to-end and focus the model on the actual task.

Chapter 4: Structural Pretraining Tasks for Generative Transformer Models

In Chapter 4, we investigate a different strategy to improve the understanding of code by incorporating structural information during the pretraining phase of transformer Language Models (LMs) without modifying their architecture or input format—the code is encoded as a regular sequence of tokens. Thereby, the chapter follows the insight that self-supervised large-scale pretraining has become a standard practice in Natural Language Processing (NLP) and that transformers benefit from denoising pretraining on extensive datasets. In the context of code, however, only few approaches have explored pretraining on code at the time of work, most of which have focused on token-level tasks (Feng et al. 2020; Ahmad et al. 2021), and even fewer have considered the structural aspects of code to construct

[1] This thesis uses the term end-to-end to describe models that are trained on the actual task, without any intermediate steps or auxiliary tasks.

denoising pretraining tasks (Lachaux et al. 2021). Specifically, this chapter explores a novel self-supervised pretraining strategy that incorporate structural information from code.

Especially, this chapter introduces a novel structural pretraining task called tree-based span selection, which selects spans for masking based on the AST of the code. This is shown to produce more challenging and contextually rich training examples than traditional random short-span masking. The chapter also extends and improves the structural pretraining task identifier deobfuscation (Lachaux et al. 2021) and trains the first multi-task LM using this task across 16 different programming languages. We call the resulting LM SyntaxPT (see Figure 1.1) and demonstrate that it outperforms the Rst model from the previous chapter on code summarization. Additionally, the chapter presents a technique called tree-based file truncation for truncating long code files to a fixed size, while preserving contextual information. Overall, the key contributions of this chapter are:

- Introduce the novel self-supervised tree-based span selection pretraining task.
- Extend and improve the pretraining task identifier deobfuscation and applying it across multiple programming languages.
- Present the tensortree library that enables developers to work with tree structures in PyTorch efficiently, which serves as a basis for implementing the above pretraining tasks. It has been open-sourced and is available on GitHub (**Villmow** 2021).
- Develop a novel multi-task pretraining approach that combines structural and regular pretraining tasks, and is trained in a self-supervised manner on unimodal code data, which we call SyntaxPT.
- Demonstrate that the structural pretraining tasks lead to better code understanding capabilities compared to regular (token-level) tasks, which we measure on five CodeXglue benchmarks (Lu et al. 2021). Additionally, SyntaxPT outperforms the state-of-the-art on four of them, including the Rst model from the previous chapter.

Overall, this chapter shows that incorporating structural information during pretraining improves the model's understanding, leading to an improved performance on downstream tasks. That's why the SyntaxPT model is used in the following chapters for code retrieval and identifier quality assessment tasks.

Chapter 5: Contrastive Pretraining for Contextualized Code Search

Code reuse is a fundamental aspect of software development, and code retrieval is a key task in this context, which is the focus of Chapter 5. In particular, we aim to develop a

model that can retrieve relevant code snippets based on a given partial coding context (e.g., the current file in the editor), with a specific position-of-interest (e.g., the cursor position). This task is called Contextualized Code Search (CCS) (Mukherjee et al. 2020), and the task's setup is similar to code autocompletion, but it rather aims to *retrieve* code segments from a codebase that implement the missing functionality instead of *synthesizing* the missing code from an LM's weights. The chapter presents a novel self-supervised approach to train CCS retrievers without the need for labeled data, using contrastive learning on context-target pairs created by erasing random blocks of code from random code contexts, where the erased block is used as the retrieval target. While this task has been effectively used in NLP to train retrievers for question-answering tasks (Lee et al. 2019), the chapter demonstrates that simply applying this trick to source code does not lead to good retrieval performance for CCS and performs worse than traditional keyword-based retrieval methods, such as BM25 (Robertson and Zaragoza 2009). This is due to trivial patterns the retriever can exploit during training. To address this limitation the chapter introduces novel deleaking steps, that remove such patterns from the training pairs. The chapter applies this pretraining strategy to the SyntaxPT model from Chapter 4 and creates a novel CCS retriever called SyntaxPT-CCS (see Figure 1.1). Also, the chapter presents a manually curated dataset based on aligned code clones called Cocos (COntextualized COde Search Dataset) for evaluating CCS models. In summary, the key contributions of this chapter are:

- Introduce the Cocos dataset, to directly evaluate Contextualized Code Search (CCS) models, based on annotated code clones. Made publically available **Villmow** (2022)[2].

- Present a novel self-supervised approach to contextualized code search using Cloze task-based contrastive pretraining on source code, along with deleaking steps to improve retrieval performance.

- Demonstrate that the proposed pretraining approach achieves state-of-the-art performance on code retrieval tasks and improves performance on encoder-based code understanding tasks such as defect detection and code clone detection, outperforming previous state-of-the-art results.

Overall, this chapter extends the application of the models developed in previous chapters to practical code retrieval tasks. From a developer's perspective, the model can be used for opportunistic reuse searching for code snippets based on the given code context.

[2] https://github.com/villmow/coling-cocos

1.1.2 Part II: Applications

Part II of this dissertation will shift the focus from the design of machine learning models and training regimes to practical applications of the developed models. It is important to note that the evaluations in Part I primarily aimed to measure the capabilities of the models for the respective task through standard machine learning benchmarks. This is common practice in ML research and important for optimizing ML pipelines. Nonetheless, such benchmarks provide only a narrow view of the model's practical utility, which depends on other factors such as user behavior and preferences. Part II explores practical applications of the models and techniques developed in Part I, in which the SyntaxPT and SyntaxPT-CCS are used in zero-shot settings to solve two practical software development problems.

Chapter 6: Evaluating Contextualized Code Search in Practical User Studies

In Chapter 6, we want to assess the real-world applicability of the proposed code retrieval model from Chapter 5 in practical software development scenarios. To this end, the chapter introduces CodeBuddy, a prototype application for CCS that allows developers to interactively search for relevant code snippets based on their current coding context. It uses the SyntaxPT-CCS model from Chapter 5 to retrieve code snippets from a codebase in a zero-shot setting. To adapt search to practical user queries—which may be incomplete or imprecise— this chapter enhances the self-supervised pretraining approach from Chapter 5 to improve the model's usability and robustness in handling real user input. It also introduces an indexing, and a post-processing strategy to filter redundant results from SyntaxPT-CCS. To evaluate CodeBuddy, we conduct two user studies: a controlled experiment with 41 computer science students working on programming exercises (Study A), and a case study with four professional software developers from the AOE GmbH in Wiesbaden using CodeBuddy in their regular work activities (Study B). Overall, the key contributions of this chapter are:

- Enhance the self-supervised pretraining approach from Chapter 5 to improve usability and robustness for end-users.
- Develop a novel post-processing strategy to filter redundant code snippets from the retrieval results.
- Present CodeBuddy, the first tool for interactive contextualized code search. The software is made publically available under **Villmow** (2024)[3].
- Evaluate CodeBuddy through two user studies, and demonstrate its practical utility and impact on developer efficiency.

[3] https://github.com/villmow/codebuddy

This chapter applies the self-supervised CCS strategy from Chapter 5 into a practical tool that can assist developers with code reuse. It studies whether this tool enhances their productivity and showcases potential use cases in real-world software development.

Chapter 7: Spotting Identifiers that Violate Naming Guidelines

In Chapter 7, we explore another practical application of the SyntaxPT model and investigate its ability to assess the quality of identifiers in source code. High-quality identifiers are important for code readability and maintainability, and thus we want to assist developers in writing good identifiers. The chapter proposes a novel self-supervised approach that assesses whether LMs can detect violations of established identifier naming guidelines. To this end, it introduces the first dataset for assessing the quality of identifiers based on coding guidelines, which consists of over 6000 dense annotations across 28 common naming guidelines. The chapter proposes and evaluates four different self-supervised scoring functions (three generative and one discriminative) to extract quality scores for identifiers from an LM. The key contributions of this chapter are:

- Create a dataset for evaluating identifier quality based on established coding guidelines. Made publically available under **Villmow** et al. (2023a)[4].
- Propose and evaluate novel scoring functions for assessing identifier quality using language models.
- Demonstration that the SyntaxPT model outperforms other state-of-the-art language models on this task, including larger models like InCoder.
- Provide an in-depth analysis of guideline-specific performance and discusses challenges in detecting certain types of guideline violations.

This chapter demonstrates the potential of LMs to aid in code quality assurance by identifying poor naming practices.

1.2 Publications

The contributions of this thesis have been published in the following peer-reviewed publications in ascending date order[5] and Page 279 provides my individual contributions to each publication:

- **Johannes Villmow**, Adrian Ulges, and Ulrich Schwanecke (2021b). A Structural Transformer with Relative Positions in Trees for Code-to-Sequence Tasks. In *International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021*. IEEE, pp. 1–10 (Chapter 3)

[4] https://zenodo.org/records/7612762
[5] A star (∗) denotes shared first authorship.

- **Johannes Villmow**, Viola Campos, Adrian Ulges, and Ulrich Schwanecke (2022). Addressing Leakage in Self-Supervised Contextualized Code Retrieval. In *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*. International Committee on Computational Linguistics, pp. 1006–1013 (Chapters 4 and 5)

- **Johannes Villmow***, Viola Campos*, Jean Petry, Amine Abbad Andaloussi, Adrian Ulges, and Barbara Weber (2023b). How Well Can Masked Language Models Spot Identifiers That Violate Naming Guidelines? In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*. IEEE, pp. 131–142 (Chapters 4 and 7)

- **Johannes Villmow**, Adrian Ulges, and Ulrich Schwanecke (2024). Evaluating Contextualized Code Search in Practical User Studies. In *INFORMATIK 2024, Wiesbaden, Germany, 24. September – 26. September 2024*. Vol. 352. LNI. GI, pp. 1393–1403. ISBN: 978-3-88579-746-3 (Chapter 6)

Additionally, the following peer-reviewed publications with me as a (co-)author have been published in the context of my doctoral studies, but have not been included in this thesis:

- **Johannes Villmow**, Marco Wrzalik, and Dirk Krechel (2018). Automatic Keyphrase Extraction Using Recurrent Neural Networks. In *Machine Learning and Data Mining in Pattern Recognition - 14th International Conference, MLDM 2018, New York, NY, USA, July 15-19, 2018, Proceedings, Part II*. vol. 10935. Lecture Notes in Computer Science. Springer, pp. 210–217

- Haseeb Shah*, **Johannes Villmow***, Adrian Ulges, Ulrich Schwanecke, and Faisal Shafait (2019). An Open-World Extension to Knowledge Graph Completion Models. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, pp. 3044–3051

- Haseeb Shah, **Johannes Villmow**, and Adrian Ulges (2020). Relation Specific Transformations for Open World Knowledge Graph Completion. In *Proceedings of the Graph-based Methods for Natural Language Processing (TextGraphs)*. Barcelona, Spain (Online): Association for Computational Linguistics, pp. 79–84

- Felix Binder, **Johannes Villmow**, and Adrian Ulges (2020). Bidirectional Transformer Language Models for Smart Autocompletion of Source Code. In *50. Jahrestagung der Gesellschaft für Informatik, INFORMATIK 2020 - Back to the Future, Karlsruhe, Germany, 28. September - 2. Oktober 2020*. Vol. P-307. LNI. GI, pp. 915–922

- **Johannes Villmow**, Jonas Depoix, and Adrian Ulges (2021a). ConTest: A Unit Test Completion Benchmark featuring Context. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Online: Association for Computational Linguistics, pp. 17–25

- Marco Wrzalik, Julian Eversheim, **Johannes Villmow**, Adrian Ulges, Dirk Krechel, Sven Spieckermann, and Robert Forstner (2023). Value Stream Repair Using Graph Structure Learning. In *Advances and Trends in Artificial Intelligence. Theory and Applications - 36th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2023, Shanghai, China, July 19-22, 2023, Proceedings, Part II*. vol. 13926. Lecture Notes in Computer Science. Springer, pp. 15–32

1.3 Methodology

The research methodology of this thesis is based on the principles of Artificial Intelligence (AI)-driven research for information systems, which is a combination of behavioral and design-oriented research. It follows the *Design Science Research* methodology (Hevner et al. 2004), which is a problem-solving paradigm that aims to create innovative solutions to real-world problems. Hevner et al. (2004) state that the goal and outcome of design science research is to develop and evaluate *artifacts*, for example, algorithms, models, prototypes, or systems, that improve the understanding of a given problem domain and provide solutions to problems in that domain. It acknowledges that the development of artifacts is an iterative process that consists of several phases, including problem identification, solution design, rigorous evaluation, and communication, that can be repeated until a satisfactory solution is found.

In Part I, the focus is on developing new models and techniques using standard machine learning research methodology (Goodfellow et al. 2016, p. 421). This involves identifying research gaps through related work; iteratively developing, adapting, and optimizing models from the literature; and rigorously evaluating them on standard datasets and benchmarks using established performance metrics. Additionally, ablation studies are conducted when feasible to assess the impact of various model components. The results of this process are communicated to the research community and also approved through peer-reviewed publications.

On the other hand, Part II applies the artifacts from the previous part (i.e., the developed models) to real-world problems. In this case, the artifact takes the form of software tools that are evaluated in practical settings. Their evaluation includes user studies to assess the utility of the models, as well as qualitative and quantitative analyses to measure their performance in real-world scenarios. Chapter 6 provides a more detailed description of

how the design science research methodology is applied to iteratively develop and evaluate the CodeBuddy prototype.

Programs must be written for people to read, and only incidentally for machines to execute.

— Abelson and Sussman 1985, p. xxii

2 Fundamentals

This chapter introduces the fundamental concepts and techniques that are essential for understanding the subsequent chapters. Most of it is based on the following books:

- Ian J. Goodfellow et al. (2016). Deep Learning. Adaptive computation and machine learning. MIT Press. ISBN: 978-0-262-03561-3
- Dan Jurafsky and James H. Martin (2009). Speech and Language Processing: an Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 2nd Edition. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International. ISBN: 9780135041963
- Christopher D. Manning et al. (2008). Introduction to Information Retrieval. Cambridge University Press. ISBN: 978-0-521-86571-5

2.1 Text and Code Processing and Representations

Text and code processing involves converting raw text and code data into a format that can be used as input to machine learning models. This process typically starts with tokenization, constructing a vocabulary construction, and the use of various representations such as subword representations and Abstract Syntax Trees (ASTs) and Concrete Syntax Trees (CSTs).

2.1.1 Tokenization and Vocabulary

Tokenization is the process of breaking a sequence of characters into smaller units, called tokens. In Natural Language Processing (NLP), tokens are typically words or subwords

```python
def fib(n):
    if n <= 1: return n
return fib(n - 1) + fib(n - 2)
```

Figure 2.1: A Python function that computes the n-th Fibonacci number.

(i.e., parts of a word), but they can also be punctuation marks or special characters. In code processing, tokens are typically programming language keywords, identifiers, and subwords of identifiers. Tokenization is typically the first step in processing text and code data for machine learning models. Consider the code snippet in Figure 2.1. A tokenization process could split the code into the following tokens: `def`, `fib`, `(`, `n`, `)`, `:`, `INDENT`, `if`, . . ., `)`.

For a sequence of characters c, the tokenization process produces a sequence of tokens $w = (w^{(1)}, w^{(2)}, \ldots, w^{(n)})$, where each token $w^{(i)}$ is a substring of c. We can build a vocabulary, denoted as $\mathbb{V}$, by collecting the set of unique tokens in a dataset. For the ease of notation—even though the vocabulary is a set—this work defines it to be sorted, so that it can be indexed like a sequence, i.e., $\mathbb{V} = \{v^{(1)}, v^{(2)}, \ldots, v^{(|\mathbb{V}|)}\}$, where $\mathbb{V}^{(j)} = v^{(j)}$ is the j-th token in the vocabulary. Let $\text{vocab}: \mathbb{V} \to \{1, \ldots, |\mathbb{V}|\}$ be a function that maps a token t to its index in the vocabulary, i.e.,

$$\text{vocab}(t) = j \quad \text{if} \quad t = v^{(j)}. \tag{2.1}$$

The vocabulary is used to map the sequence of token-strings w to a numerical vector $\boldsymbol{x} = (x^{(1)}, x^{(2)}, \ldots, x^{(n)})$ that can be used as input to machine learning models, where each element $x^{(i)}$ is the index of the token $w^{(i)}$ in the vocabulary, i.e., $x^{(i)} = \text{vocab}(w^{(i)})$.

In addition to the tokens in the dataset, the vocabulary also contains some special tokens that do not correspond to actual words or code tokens in the input data:

- `[UNK]`: Unknown token used to represent tokens that are not in the vocabulary.
- `[PAD]`: Padding token used to fill sequences to a certain length, for example, when batching multiple sequences of different lengths. This token is ignored when computing the loss.
- `[CLS]`: Classification token used to represent the start of a sequence.
- `[EOS]`: End-of-sequence token used to represent the end of a sequence.
- `[MASK]`: Mask token used to represent parts that have been hidden.

Throughout this thesis, notations such as $x^{(mask)}$ or $y^{(eos)}$ are used to denote such special tokens, i.e., $x^{(mask)} = \text{vocab}(\texttt{[MASK]})$.

Jurafsky and Martin (2009, p. 19) state that there are two types of tokenization algorithms: top-down and bottom-up tokenization. Top-down tokenization defines a set of rules that split text into tokens, while bottom-up tokenization learns the tokenization rules from the data. Bottom-up tokenization is also referred to as subword tokenization. Throughout this thesis, both types are used in combination to tokenize source code.

Top-down Tokenization

Top-down or rule-based tokenization consists of a set of rules that define how to split text into tokens, similar to regular expressions. For natural language this approach can be challenging, as the rules are often language-specific and hard to generalize. Think of the ambiguity between apostrophes in English, which can be used to indicate possession or contraction (e.g., *John's car* vs. *John's going to the store*). This makes it difficult to define a set of rules that can accurately tokenize text in natural language. However, the opposite is true for source code. Since source code is well-defined by a context-free grammar, the grammar already defines the necessary rules for tokenization. A rule-based tokenization strategy has the advantage that the parser generates a syntax tree that represents the syntactic structure of the code. The syntax tree for the code snippet in Figure 2.1 is shown in Figure 2.2. These representations for code will be introduced in Section 2.1.2. In addition to these structural properties, source code also contains natural language elements, e.g., identifier names, strings, and comments. Tokenizing these elements can be as complicated as in Natural Language Processing (NLP).

The design of the tokenization algorithm strongly influences the amount of information that is retained in the token sequence. For example, in the Java programming language, whitespace is not important, and can be removed without losing information (except for string literals). In Python, however, whitespace is used to indicate the structure of the code, and removing it would result in a loss of information. Consider the code snippet `int studentCount = 5;`, that can be tokenized with a parser into the following tokens: `int`, `studentCount`, `=`, `5`, `;`. Some approaches do not retain the whitespace tokens, which shortens the tokenized sequence, but makes the tokenization irreversible—the original code cannot be reconstructed from the token sequence. For languages like Python, special indent and dedent tokens can be used to indicate the structure of the code.

Top-down approaches have the drawback, that the size of $\mathbb{V}$ "for a text goes up significantly faster than the square root of its length in words" (Jurafsky and Martin 2009, p. 15). Every misspelling of a word will result in a new token in the vocabulary. When training machine learning models for text and code processing, a major computational factor is the size of the vocabulary. For example, in order to predict the next token in a sequence, the model must compute a probability distribution over the entire vocabulary with the softmax function. Large vocabulary sizes increase the complexity of this task. Hence, the vocabulary is

often limited to the most frequent tokens in the dataset. But then out-of-vocabulary (OOV) issues arise at inference time, when the model encounters tokens not seen during training (e.g., that were too infrequent). Typically, an unknown token at inference time is represented with a specific marker $x^{(unk)}$. Various approaches, such as copy mechanisms in sequence-to-sequence pointer networks (Vinyals et al. 2015) that allow models to copy tokens from the input sequence, have been developed to address this issue. A more recent and widely adopted approach is to use subword tokenization, which is discussed in the following section.

Subword Tokenization

Subword tokenization is a bottom-up approach that learns the tokenization rules from the data. With subword tokenization, words are broken down into smaller units, called subwords, that can capture morphological information and handle OOV words. For example `studentCount` becomes `student` and `@@Count`, where `@@` denotes that the token is a continuation of the previous token. The maximum amount of subword tokens is a hyperparameter that can be adjusted to control the size of the vocabulary. Typically, it is much smaller than the number of unique words in the dataset and ranges between 32k tokens, used in CodeT5 (Wang et al. 2021b), and 250k tokens, used in XLM-R (Conneau et al. 2020). The smallest base-units can be the individual characters of the alphabet, or even better the 128 possible values of a byte. When a word is so uncommon, that none of the bigger subwords match, the word can be at least segmented in a sequence of characters. However, this might not be enough given that Unicode in version 16.0 contains more than 150k characters. To address this issue, many approaches use byte-level base-units (Wang et al. 2020) that segment also Unicode characters through a sequence of byte tokens. This effectively solves the OOV problem.

Different subword tokenization algorithms have been developed, including *Byte Pair Encoding (BPE)* (Sennrich et al. 2016), *Unigram Language Model* (Kudo 2018), and the proprietary *WordPiece* algorithm (Devlin et al. 2019). Throughout this thesis, all of these algorithms are referred to as BPE. In Chapter 3 a subword tokenization model is trained with the Byte Pair Encoding (BPE) algorithm proposed by Sennrich et al. (2016), while in Chapter 4 a subword tokenization model is trained with the Unigram Language Model from Kudo (2018). These two types of models are now explained in detail.

Byte Pair Encoding The earliest approach to subword representations and its namesake is the Byte Pair Encoding (BPE) algorithm, a data compression algorithm proposed by Gage (1994) and applied to machine learning by Sennrich et al. (2016). The BPE algorithm is trained by learning merge operations. Starting with a vocabulary of base-units, the algorithm iteratively merges adjacent tokens into new tokens based on their frequency in the dataset, until a predefined vocabulary size is reached. The same merge

operations can be applied at inference time to tokenize new data. The resulting vocabulary of merged tokens contains subwords that can be used to represent words in the dataset. One drawback of the algorithm proposed by Sennrich et al. (2016) is that it requires an initial tokenization on whitespace, which makes the tokenization process irreversible. A byte-level version of this algorithm is used in the GPT-2 model (Radford et al. 2019).

Unigram Language Model Kudo (2018) proposed a probabilistic algorithm that assigns a probability to each subword token in the vocabulary, called the unigram language model. It is implemented in the SentencePiece library (Kudo and Richardson 2018). It is formulated as an "entropy encoder that minimizes the total code length for the text" (Kudo 2018, p. 69). To this end the approach first gathers a large vocabulary of subword tokens, that exceeds the desired vocabulary size by far, and then iteratively removes the least probable tokens with an expectation-maximization algorithm until the desired size is reached (Mielke et al. 2021). The tokenization model can output different segmentations for the same input, because of the probabilistic nature of the model, which can potentially improve robustness of the model—similar to dropout. However, this is rarely used, because typically the training data is encoded in a preprocessing step for faster loading times during training. In SentencePiece, whitespace is considered a token, so no pre-tokenization is required, and the tokenization is *reversible*. The unigram language model is used in the T5 model (Raffel et al. 2020). Even though the SentencePiece library implements both subword tokenization algorithms, when referring to the unigram language model, the term SentencePiece is used throughout this thesis.

2.1.2 Representations of Source Code

In machine learning on source code, the code is commonly represented in various forms. For example, Li et al. (2023) follow common practice in NLP and use a sequence of code tokens as input to a model. Alon et al. (2019b) and Alon et al. (2019a) use ASTs to represent source code (see Figure 2.2), while Guo et al. (2021) use program dependency graphs (Ferrante et al. 1987), that contain information about the control and data flow of the program.

Abstract Syntax Trees

Abstract Syntax Trees (ASTs) are widely used throughout the machine learning on code community. They have found use in applications for accurate source code differencing (Falleri et al. 2014), automatic program repair (Weimer et al. 2009), source code summarization (Alon et al. 2019a), and source code search (Paul and Prakash 1994). An AST is a tree representation of the syntactic structure of source code, which results from parsing the source code with a context-free grammar. ASTs are constructed by parsers like javalang (Thunes 2023), a Java parser implemented in Python, or tree-sitter (Brunsfeld

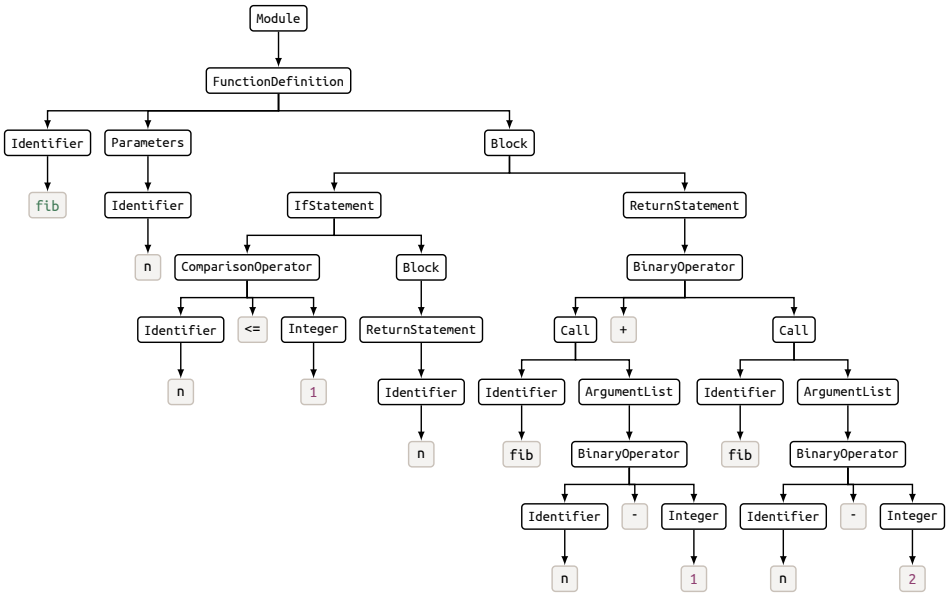


Figure 2.2: The AST of the Fibonacci function shown in Figure 2.1.

2023), which supports several programming languages. The grammar's production rules determine the AST's structure: nonterminal nodes represent grammar rules and terminal nodes represent source code elements. The AST excludes certain code elements which are captured by its structure, such as keywords (`def`, `return`), comments, and punctuation (brackets, colons).

Formally an AST is a rooted tree with an unlimited degree. Given the ubiquity of rooted trees in computer science, a formal definition of a tree is omitted in this thesis. Instead, a short informal definition is given here. A tree has a single node called the root node, which is the only node without a parent. All other nodes have exactly one parent. Because the degree is unlimited, each node can have an arbitrary number of children, including zero. Nodes without children are called *leaf* or *terminal nodes*, while a node that has at least one child is called a *nonterminal node*. Every node has a label, which is a string that describes the node's role in the tree. For nonterminal nodes the label is a value from the set of nonterminal labels $\mathbb{L}$, while for terminal nodes the label can be an arbitrary string.

For instance, in Figure 2.2 the AST of the `fib` function from Figure 2.1 is shown. In the tree, nonterminal nodes such as `Module`, `FunctionDefinition` and `ReturnStatement` correspond to grammar rules, while terminals such as `fib` and `n` represent source code tokens. Typically, ASTs are language-specific, since the nonterminal nodes originate from the grammar of the programming language and the grammar rules are not universal across

Language	Unique Nonterminals	Files	Nonterminals per File
C	101	2,242,379	1036.0
C#	218	2,843,642	717.5
C++	165	3,734,357	1014.4
CSS	52	349,525	816.8
Go	93	1,759,600	1067.8
Haskell	180	114,311	1005.9
Java	123	7,345,753	720.8
JavaScript	109	4,471,689	720.7
Julia	82	34,403	522.8
OCaml	193	55,838	1226.3
PHP	141	2,206,063	616.5
Python	100	3,016,545	938.7
Ruby	112	1,068,668	459.2
Rust	152	366,891	1111.9
Scala	100	273,822	506.8
TypeScript	166	2,299,964	504.0

Table 2.1: Statistics on the number of nonterminal labels $|\mathbb{L}|$ in tree-sitter ASTs and CSTs for various programming languages. The data was collected by analyzing 237k open-source repositories on GitHub.

languages[1]. For example in tree-sitter, the nonterminal for a function in Python is called `FunctionDefinition`, while for Java it is called `MethodDeclaration`. Table 2.1 shows the number of unique nonterminal labels in the AST and CST for various programming languages, as extracted from 237k open-source repositories on GitHub.

For each node i in a tree, there is a unique path from the root to that node. This path is represented as $(i_1, \ldots, i_u)$, where $i_1 = 1$ is the root, and $i_u = i$ is the node in question, and for all $l \in \{2, \ldots, u\}$, i_{l-1} is the parent of node i_l. The depth of the node i is equal to the number of nodes u on this path:

$$\text{depth}(i) = u \tag{2.2}$$

The set of ancestors for node i includes all nodes along this unique path. Note that node i is part of its own ancestor set:

$$\text{ancestors}(i) = \{i_1, \ldots, i_u\} \tag{2.3}$$

However, i is excluded from the set of descendants of node i, which comprises all nodes for which i is an ancestor:

$$\text{descendants}(i) = \{j \mid i \in \text{ancestors}(j) \wedge\ i \neq j\} \tag{2.4}$$

[1]While most ASTs are language-specific, some approaches, such as Babelfish (2020), try to standardize AST representations across programming languages. Therefore, they define a universal AST format that works on many source code languages. However, these are not used in this thesis.

The Lowest Common Ancestor (LCA) of two nodes i and j is defined as the ancestor a common to both and having the greatest depth.

$$\text{lca}(i, j) = \underset{a \in \text{ancestors}(i) \cap \text{ancestors}(j)}{\arg\max} \text{depth}(a) \tag{2.5}$$

In an AST, the LCA tells us to which syntactical construct both nodes belong.

Concrete Syntax Trees

Concrete Syntax Trees (CSTs) contain every token in the source code as a terminal node, unlike ASTs. Comparing the AST in Figure 2.2 with the CST in Figure 2.3 shows that, although they share identical nonterminal node structures, the CST has additional terminal nodes—highlighted in orange outlines—that were omitted in the AST. This results in a higher node or token count than the AST. Parsers like tree-sitter generate CSTs without whitespace. In this thesis, however, CSTs are adapted to explicitly include whitespace.

Because the CST includes all tokens and whitespace, it can be easily reverted to source code by concatenating the terminal nodes. This is useful for source code manipulation, including comment removal, variable renaming, and code reformatting.

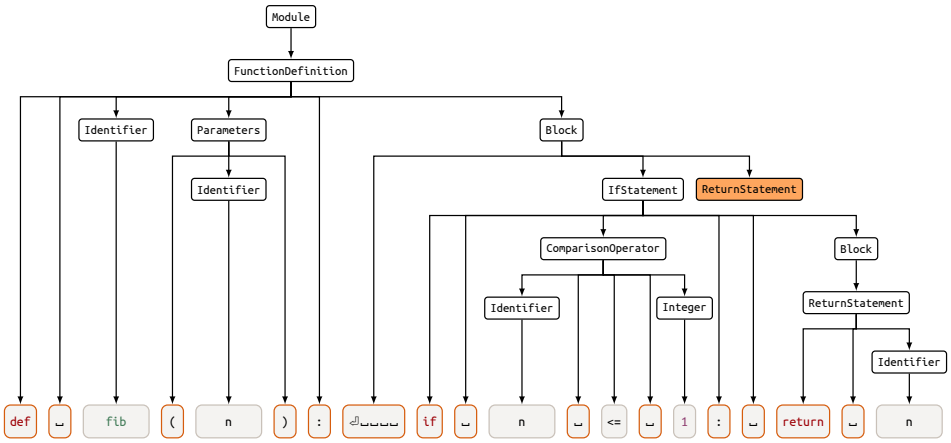


Figure 2.3: The Concrete Syntax Tree (CST) of the Fibonacci function shown in Figure 2.1. This figure shows a partial CST, truncated at the orange `ReturnStatement` node for brevity. New tokens compared to the AST from Figure 2.2 are highlighted with orange outlines. Space characters are visualized by the symbol ␣ and line breaks by ⏎. Note that all leaves together make up the original source code.

2.2 Machine Learning Fundamentals

Machine Learning (ML) is a powerful approach to solve tasks by learning from data instead of manually designing algorithms and decision logic. Machine Learning (ML) algorithms (also called models) have been found in the last 10 years to be excellent at recognizing patterns and making decisions in many domains. Problems that were previously considered

too complex to solve with traditional programming methods, such as machine translation, can now be solved with machine learning, and in particular deep learning models. ML models need to be trained on a dataset that contains examples of the task they are supposed to solve before they can make predictions on new data. In essence, ML models define a parametric function $f(\boldsymbol{x}; \boldsymbol{\theta})$ that maps an arbitrary input $\boldsymbol{x}$ to an output $\boldsymbol{y}$, whereas the function's parameters $\boldsymbol{\theta}$ are learned from the training data. In this thesis the input to all models is a sequence or vector of token indices $\boldsymbol{x} \in \{1, \ldots, |\mathbb{V}|\}^n$ produced by the tokenization process described in Section 2.1.1.

2.2.1 Training

The models in this thesis are neural networks, in particular deep learning models, which are typically trained using maximum likelihood (Goodfellow et al. 2016, p. 173). The model function outputs a probability distribution and the *cross-entropy loss* can be used to train the model.

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}) \tag{2.6}$$

For an input $\boldsymbol{x}$, typically described by a vector, and a corresponding desired output $\boldsymbol{y}$, the goal is to find a maximum likelihood estimator so that the model distribution closely matches the empirical distribution of the training data $\hat{p}_{\text{data}}$ for parameters $\boldsymbol{\theta}$.

$$\hat{\boldsymbol{\theta}}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{(\text{x},\text{y})\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}) \tag{2.7}$$

Finding this maximum likelihood estimator is equivalent to *minimizing* the Kullback-Leibler divergence—which measures how different two probability distributions are—between the model and the empirical distribution in a loss function. In practice, this is commonly referred to as cross-entropy or negative log-likelihood loss.

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}) &= D_{\text{KL}}(\hat{p}_{\text{data}} \parallel p_{\text{model}}) \\ &= \mathbb{E}_{(\text{x},\text{y})\sim\hat{p}_{\text{data}}} \left[\log \hat{p}_{\text{data}}(\boldsymbol{y} \mid \boldsymbol{x}) - \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})\right] \\ &\equiv -\mathbb{E}_{(\text{x},\text{y})\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}) \end{aligned} \tag{2.8}$$

One advantage of this approach is that designing a model that outputs $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$ automatically defines a loss function that can be directly used to train the model (Goodfellow et al. 2016). To this end, mini-batch stochastic gradient descent is used to approximate the expected value of the loss function by sampling mini-batches of size N from the training data. Here a batch of paired examples $(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), \ldots, (\boldsymbol{x}^{(N)}, \boldsymbol{y}^{(N)})$ is drawn from the training data.

$$\mathcal{L}(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} - \log p_{\text{model}}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}) \tag{2.9}$$

For brevity, the parameters $\boldsymbol{\theta}$ and the expected value of the loss are omitted in the subsequent sections and the negative log-likelihood loss is denoted as:

$$\mathcal{L}(\boldsymbol{\theta}) = -\log p(\boldsymbol{y} \mid \boldsymbol{x}) \tag{2.10}$$

Optimization and Stochastic Gradient Descent

Optimization is the process of finding the best parameters $\hat{\boldsymbol{\theta}}$ for a model that minimize the loss function $\mathcal{L}(\boldsymbol{\theta})$. To do so, gradient descent is a widely used optimization algorithm that iteratively updates the parameters in the direction of the negative gradient of the loss function with respect to the model's parameters $\nabla \mathcal{L}_{\boldsymbol{\theta}}$. The gradient is calculated via backpropagation, whereas modern frameworks such as PyTorch (Paszke et al. 2019) automate this differentiation process.

Gradient descent's most important hyperparameter, the learning rate h_{lr} controls the step size in the direction of the gradient and typically ranges between 10^{-6} and 10^{-1}. Ideally, the gradient would be computed over the entire dataset, but this is computationally prohibitive for large datasets (Goodfellow et al. 2016, p. 152). To combat this, one approximately estimates the gradient by computing it on a random mini-batch of size h_b, over which the individual per sample losses are averaged. This approach is called mini-batch stochastic gradient descent. The batch size is another hyperparameter that affects performance, and training continues until one observes *overfitting*, which is determined by monitoring the loss on a validation set.

There are many modern adaptive variants of gradient descent, such as Adam (Kingma and Ba 2015) and Adafactor (Shazeer and Stern 2018), which adapt the learning rate for each parameter based on the first and second moments of the gradient. Since Adam is widely used in practice and fairly robust with respect to the choice of hyperparameters (Goodfellow et al. 2016, p. 309) it is used as the default optimizer in this thesis. Specifically, most experiments use the AdamW variant of the Adam optimizer (Loshchilov and Hutter 2019), which decouples weight decay from the optimization steps and has been shown to improve the performance of transformers in practice (Radford et al. 2018).

Datasets and Hyperparameter Optimization

Training of deep neural networks requires large amounts of data, which is typically split into three sets: the training set, the validation set, and the test set. The training set is used to train the model, the validation set is used to tune hyperparameters, and the test set is used to measure the final model's generalization performance. Many of the benchmark datasets used in this thesis are publicly available, widely used in the research community, and come with predefined splits to ensure comparability with other models (e.g., the CodeXGlue benchmark collection).

Special attention should be paid to the dataset split in machine learning for source code. A naive split at example level may result in the same identifiers or methods appearing all over training, validation, and test sets, leading to data leakage. To circumvent this issue, datasets for machine learning on source code are typically split at a project level, so that the model's ability to generalize to new, unseen projects can be accurately measured. This thesis evaluates only on datasets that adhere to a project-level split.

As it is common practice in ML, the models in this thesis have been developed iteratively. Following Goodfellow et al. (2016, p. 421) first one or more *performance metrics* are specified, next a working pipeline is established that allows to train and evaluate models on the training and validation set. Then the models are iteratively improved by changing their architecture, the loss function, the data handling, or other hyperparameters. For example in Chapter 5, the author tried contrastive learning with a triplet margin loss before switching to the InfoNCE loss, which was found to lead to a higher validation performance. During the iterative development phases in this thesis, the author measured performance exclusively on the validation set and the test set was only used once at the end of the development process. This is important because frequent ablation on the test set increases the risk of overfitting, which can lead to overly optimistic results.

Hyperparameter Optimization Hyperparameter optimization is the process of finding the best hyperparameters for a machine learning model. Hyperparameters are parameters that are not part of the training/optimization process. For example, the aforementioned process of measuring the benefit of adding a different loss function is an example of hyperparameter optimization. Such hyperparameters, that are specific to the model architecture often include the number of layers, the number of hidden units, and the dropout rate, are typically determined only once and then kept fixed for all test runs. Hyperparameters that should be fine-tuned for each specific task and dataset include the learning rate h_{lr}, the batch size h_b, and the number of epochs h_e.

The more general architectural hyperparameters are set based on the literature or determined by the aforementioned iterative process. The more fragile hyperparameters are optimized using a hyperparameter optimization strategy called *sweep*. A sweep explores different hyperparameter configurations, keeps the best one in a trial and error fashion, and then uses the best configuration for the final evaluation. A search strategy decides which configurations to explore, for example, grid search, random search, Bayesian optimization, and evolutionary algorithms. The ones used in this thesis are grid search and Bayesian optimization. Grid search is a brute-force approach that evaluates all possible combinations of a predefined set of hyperparameters. Bayesian optimization models the hyperparameter space as a probabilistic model and iteratively samples the most promising hyperparameters. This thesis uses the implementation of the Weights and Biases platform (Biewald 2024).

Overfitting and Early Stopping One common problem in machine learning is overfitting, which occurs when the model performs well on the training set but poorly on the validation set. This is often caused by the model memorizing training data (or noise in it) or the model being too complex for the given data. Large models with many parameters, that have large capacity, are more prone to overfitting. This work closely follows the early stopping approach described by Goodfellow et al. (2016, p. 239) to prevent overfitting and reduce training time during sweeping: During training the best model checkpoint according to the validation performance is saved, and the run is stopped if its performance did not improve over three consecutive epochs.

2.2.2 *Machine Learning Tasks*

A *machine learning task* defines how the system should process an example (Goodfellow et al. 2016), i.e., the task defines the output of the model function $f(\boldsymbol{x})$, mapping the input $\boldsymbol{x}$ to an output $\boldsymbol{y}$. While the prediction target $\boldsymbol{y}$ has been abstract so far, in practice, it represents a concrete output serving a specific purpose defined by the task. For example, in regression tasks, the model predicts a continuous value; in classification tasks, it predicts a class label or a probability distribution over classes; and in sequence prediction tasks, it generates a sequence of tokens. The following sections introduce the machine learning tasks relevant to this thesis.

Classification

Classification is one of the most common machine learning tasks and is the foundation of all tasks addressed in this thesis. The goal of classification is to predict which of k classes an input belongs to. Examples in this thesis include next-token prediction in sequence prediction models (where tokens in the vocabulary are the classes), defect detection (predicting whether code contains a bug), and similarity search (determining which of k code snippets matches a query). Classification is also widely used in other domains, such as image classification (e.g., determining whether an image contains a dog, cat, or horse), sentiment analysis (classifying a text as positive or negative), and spam detection.

All ML models used in this thesis represent a function $f : \mathbb{V}^n \mapsto \mathbb{R}^k$. Typically, a softmax layer converts the model's k-dimensional output logits $\boldsymbol{z} = f(\boldsymbol{x})$ into a probability distribution:

$$p_{\text{model}}(y \mid \boldsymbol{x}) \coloneqq \text{softmax}(\boldsymbol{z}) \tag{2.11}$$

$$= \frac{\exp(z_y)}{\sum_{j=1}^{k} \exp(z_j)} \tag{2.12}$$

where $y \in \{1, \ldots, k\}$ is the true class label. Subsequently, training is done by minimizing the negative log-likelihood loss, as detailed in Section 2.2.1. Note that in this thesis, when

a classification model outputs logits instead of a probability distribution, the output is converted to one with softmax. Afterwards, the most likely class is selected using the argmax function:

$$\hat{y} = \underset{y \in \{1,\dots,k\}}{\arg\max}\ p_{\text{model}}(y \mid \boldsymbol{x}) \tag{2.13}$$

Sequence Prediction

Sequence prediction is a structured output task where the goal is to predict a sequence of tokens that depend on one another, often based on a sequence of input tokens. Most experiments in this thesis are sequence prediction tasks, e.g., machine translation, code summarization, and language modeling. Specifically, this thesis studies sequence-to-sequence models, which are a class of models that map an input sequence to an output sequence.

For an input $\boldsymbol{x} = (x^{(1)}, \dots, x^{(n)})$, the model generates an output sequence by iteratively predicting the next token $\hat{y}^{(i)} \in \mathbb{V}$ given the input and previously generated tokens $\hat{\boldsymbol{y}}^{(<i)}$. The model outputs a probability distribution over tokens in the vocabulary:

$$\hat{y}^{(i)} = \underset{w \in \mathbb{V}}{\arg\max}\, p_{\text{model}}(w \mid \hat{\boldsymbol{y}}^{(<i)}, \boldsymbol{x}) \tag{2.14}$$

Token-Selection Strategies In sequence prediction models, the decoding process predicts the next token based on the input and previously generated tokens, until the model predicts a special end-of-sequence token `[EOS]`.

- The simplest token selection strategy is *greedy decoding*, which selects the token with the highest probability (as in the last equation). However, it can lead to suboptimal results by getting stuck in local optima (Jurafsky and Martin 2009, p. 233). Also, greedy decoding has been found to lead to repetitive outputs, which can be circumvented by prohibiting the model from predicting the same n-grams multiple times in a row (Paulus et al. 2018).
- *Sampling* is another, equally efficient strategy that introduces randomness by sampling from the model's output distribution:

  $$\hat{y}^{(i)} \sim p_{\text{model}}(w \mid \hat{\boldsymbol{y}}^{(<i)}, \boldsymbol{x}) \tag{2.15}$$

  However, sampling can lead to unstable or nondeterministic results, which can be problematic for reproducibility. Variants like top-k sampling and top-p sampling control randomness by sampling from the top-k tokens or the top-p percent of the probability mass, respectively (Holtzman et al. 2020). This can make the generated sequences appear more human-like.

- *Beam search* is a more sophisticated strategy commonly used in machine translation (Jurafsky and Martin 2009, p. 276). It keeps track of the k most likely sequences and expands each by predicting the next token. This results in a tree of possible sequences, where only the k most likely sequences, for which the sum of the tokens' log-probabilities is the highest, are kept and expanded. At the end the sequence with the highest overall probability is selected. In this thesis unless otherwise stated, beam search is used with a beam width of $k = 5$, which is the most common setting (Jurafsky and Martin 2009, p. 281).

Unless otherwise stated, this thesis uses beam search for the final predictions on the test set, and greedy decoding for the predictions during validation.

Training and Loss The training data for sequence-to-sequence models consists of paired examples $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{y} = (y^{(1)}, \ldots, y^{(m)})$ is the ground truth target sequence. The loss function minimizes the average negative log-likelihood of the target tokens:

$$\mathcal{L}_{\text{seq}}(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \log p_{\text{model}}(y^{(i)} \mid \boldsymbol{y}^{(<i)}, \boldsymbol{x}) \tag{2.16}$$

It is common practice to train the model using *teacher forcing* (Williams and Zipser 1989), where the model predicting the output $y^{(i)}$ receives the sequence of previous ground truth outputs $\boldsymbol{y}^{(<i)}$ as input during training (Goodfellow et al. 2016).

Label-Smoothed Cross Entropy Loss An alternative to the cross-entropy loss is the label-smoothed cross-entropy loss, which encourages less confidence in the model and acts as a regularization technique (Goodfellow et al. 2016, p. 243). Label smoothing replaces the hard targets of the cross entropy loss with a soft distribution, in which the correct class is correct only with a probability of $1 - \epsilon$. The remaining probability mass of $\epsilon \in [0, 1]$ is spread across all other possible classes or—in our case—tokens in the vocabulary $\mathbb{V}$.

$$\begin{aligned} \mathcal{L}_{\text{seq-smooth}}(\boldsymbol{\theta}) = -\sum_{i=1}^{m} \Bigg[& (1-\epsilon) \cdot \log p_{\text{model}}(y^{(i)} \mid \boldsymbol{y}^{(<i)}, \boldsymbol{x}) \\ & + \epsilon \cdot \frac{1}{k-1} \cdot \sum_{t \in \mathbb{V}, t \neq y^{(i)}} \log p_{\text{model}}(t \mid \boldsymbol{y}^{(<i)}, \boldsymbol{x}) \Bigg] \end{aligned} \tag{2.17}$$

The term $\log p_{\text{model}}(y^{(i)} \mid \boldsymbol{y}^{(<i)}, \boldsymbol{x})$ refers to the log-probability of the actual next token $y^{(i)}$, while $\log p_{\text{model}}(t \mid \boldsymbol{y}^{(<i)}, \boldsymbol{x})$ is the log-probability for all other tokens t in the vocabulary.

Contrastive Learning

This section discusses contrastive learning, which is used in the context of Contextualized Code Search (CCS) in Chapter 5 to retrieve code snippets that complement each other and for clone detection in Section 4.5.7. Contrastive learning—sometimes also called metric learning or learning to rank—is a machine learning approach that aims to learn a relation or similarity measure between pairs of inputs. This is a broad concept that can be applied to many different tasks. For example, contrastive learning can be used to learn clone detection (document-document similarity), to train Information Retrieval (IR) systems (query-document similarity) (Karpukhin et al. 2020), but also to learn word embeddings (Mikolov et al. 2013b), to verify that two handwritten signatures are the same (Bromley et al. 1993), and to determine which pairs of nodes in a graph are most likely to be connected (Shah* et al. 2019).

This section focuses on contrastive learning in the context of IR. In this scenario a pair of two different objects forms the model input: *queries* and *documents* (the documents are also referred to as *keys* in the context of contrastive learning). Some documents are relevant to a query, while others are not. The goal is to learn mappings from queries and documents to a common latent space, in which the similarity between the query and relevant documents is higher than the similarity between the query and irrelevant documents. During inference, one first encodes all documents into representations and can then retrieve the most relevant documents by computing the similarity between the query's and documents' representations, using (approximate) nearest neighbor searches (Malkov and Yashunin 2020) for which libraries like FAISS (Douze et al. 2024) or vector databases exist (Zayarni 2023).

Similarity Functions Similarity is a measure of how much two things are alike. This is a fuzzy concept, and depends on the use case. For example, in IR, one could consider two documents similar if they share many words, however, that might not capture all aspects of similarity. For example, the words `database` and `db` are semantically similar but not identical. A common measure for the similarity of vectors is the *cosine similarity*, which measures the cosine of the angle between two equally sized vectors $\text{sim}_{\text{cosine}} : \mathbb{R}^d \times \mathbb{R}^d \mapsto [-1, 1]$. It is defined as:

$$\text{sim}_{\text{cosine}}(\boldsymbol{q}, \boldsymbol{k}) = \frac{\boldsymbol{q} \cdot \boldsymbol{k}}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{k}\|} \tag{2.18}$$

When both vectors are normalized, the cosine similarity is equivalent to the dot product of the two vectors, which can be computed efficiently using matrix multiplication. Often the term *distance* is used instead of similarity, which is simply the inverse of the similarity function, i.e., $\text{dist}_{\text{cosine}} \mapsto [0, 2]$, where $\text{dist}_{\text{cosine}}(\boldsymbol{q}, \boldsymbol{k}) = 1 - \text{sim}_{\text{cosine}}(\boldsymbol{q}, \boldsymbol{k})$. Thus, cosine distance tells us how dissimilar two vectors are. Other commonly used similarity or distance functions include the Euclidean distance (L2 distance), which measures the

straight distance between two points in space, and the Manhattan distance (L1 distance), which measures the distance between two points by summing the absolute differences of their coordinates.

Contrastive Architectures The general approach in this thesis is to learn similarity between two inputs from examples with relevance labels, i.e., the pair of inputs is either similar or dissimilar (relevant or irrelevant). To this end, the model is trained to maximize the similarity between similar pairs and minimize the similarity between dissimilar pairs. Contrastive models typically utilize a joint embedding architecture (Bromley et al. 1993; He et al. 2020; Chen et al. 2020; Bardes et al. 2022), which is often referred to as a siamese network (Bromley et al. 1993) (when the two inputs are encoded by the same model) or as a bi-encoder architecture. A bi-encoder architecture consists of two machine learning models that encode queries $\boldsymbol{x}$ and keys $\boldsymbol{y}$ into dense representations $\boldsymbol{q}=f(\boldsymbol{x};\boldsymbol{\theta})$ and $\boldsymbol{k}=f'(\boldsymbol{y};\boldsymbol{\theta}')$, respectively. The resulting representations $\boldsymbol{q}, \boldsymbol{k} \in \mathbb{R}^d$ are called *sequence embeddings* and are subsequently used to measure similarity. Several approaches adopt shared weights between the two encoders (Gao et al. 2021b; Bardes et al. 2022), i.e., $\boldsymbol{\theta} = \boldsymbol{\theta}'$, which leads to a siamese network architecture (Bromley et al. 1993) where both sequences are encoded by the same encoder. This practice reduces the overall amount of parameters and is utilized throughout this thesis. In some approaches the sentence embeddings $\boldsymbol{q}, \boldsymbol{k}$ are projected to a different dimensionality d' by projectors $g(\boldsymbol{q};\boldsymbol{\theta}_g)$ and $g'(\boldsymbol{k};\boldsymbol{\theta}_{g'})$ (Bardes et al. 2022) before similarity computation. A lower dimensionality speeds up the retrieval process, since embeddings are compared against millions of other embeddings, however, it can also reduce the quality of the embeddings. Having different projectors for context and query can also be used to introduce asymmetry to the siamese network.

Contrastive Loss Consider a set of K documents $\{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(K)}\}$ and a single query $\boldsymbol{x}$. One of the documents $\boldsymbol{y}^{\oplus}$ is relevant to the query, while the remaining $K-1$ are irrelevant. These are called *negative examples*. The corresponding sequence embeddings are denoted with $\boldsymbol{q}$ for the query, and $\boldsymbol{k}^{(i)}$ for the documents, from which $\boldsymbol{k}^{\oplus}$ is the embedding for the relevant document. A contrastive loss is low when the similarity between the query and the relevant document is high and the similarity between the query and irrelevant documents is low. This thesis uses the following InfoNCE loss proposed by Oord et al. (2018) for all contrastive learning tasks:

$$\mathcal{L}_{\text{InfoNCE}}(\boldsymbol{\theta}) = -\log \frac{\exp(\text{sim}_{\text{cosine}}(\boldsymbol{q}, \boldsymbol{k}^{\oplus})/\tau)}{\sum_{i=1}^{K} \exp(\text{sim}_{\text{cosine}}(\boldsymbol{q}, \boldsymbol{k}^{(i)})/\tau)} \tag{2.19}$$

This is essentially a normalized temperature scaled cross entropy loss, where τ=0.1 is a temperature hyperparameter (Chen et al. 2020). The loss is minimized when $\boldsymbol{q}$ and $\boldsymbol{k}^{\oplus}$ are identical and $\boldsymbol{q}$ is dissimilar to all negative examples[2].

Negative Sampling Minimizing the loss function enforces the model to perform pair matching, i.e., identifying the correct pair among a set of pairs, in which only one pair is relevant. The difficulty of this task directly influences the extent to which the model must semantically understand the given content. This correlation has been confirmed by recent work, which demonstrated that the quality of negative examples significantly impacts overall performance (Chen et al. 2020; Ren et al. 2021), and corresponds to the intuition that increasing the number of negative examples increases the difficulty of the task. One can manually choose the most difficult negative examples from a dataset (which could potentially be mislabeled), or learn complementary rankings by choosing negative examples another model favors (Wrzalik and Krechel 2021). However, most common because of its simplicity is to use random negative sampling, for which the implementation is straightforward: Since batches are constructed randomly from the dataset, the remaining examples from the batch that are not relevant to the query are used as negative examples. This is called *in-batch negative sampling* and is used in this thesis[3]. It is possible to construct pairs not only using (query, document) pairs, but also (query, query) and (document, document) pairs, which additionally increases the number of negative examples (Chen et al. 2020) and is used in this thesis. This process is done for all positive pairs in the batch and the individual losses are averaged.

2.2.3 Evaluation Metrics

Evaluation metrics are used to measure the performance of machine learning models on specific tasks. The type of the task, but also the dataset determine which metrics can be used. For example, classification tasks typically report F1-score, while sequence prediction tasks use BLEU (Bilingual Evaluation Understudy). Balanced datasets can be evaluated using accuracy, while imbalanced datasets are better evaluated using precision and recall. Performance of IR models can be assessed with the Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Normalized Discounted Cumulative Gain (nDCG). The following sections introduce the evaluation metrics used in this thesis.

[2] Many other loss functions exist for contrastive learning, such as well known margin-based triplet losses or the more recent VICReg loss (Bardes et al. 2022). However, many self-supervised approaches to contrastive learning have demonstrated that the InfoNCE loss achieves a strong performance (Sohn 2016; He et al. 2020; Gao et al. 2021b). Hence, it is used in this thesis.

[3] Note that for a batch with h_b pairs, this must not mean that there are $h_b - 1$ negative documents, as some other examples in the batch might also be relevant to the query and thus are excluded from the negative examples for this query.

Precision, Recall, and F1-score

The most commonly used evaluation metrics are precision, recall, and F1-score. In binary classification tasks, TP, TN, FP, and FN represent true positives (correctly predicted positive samples), true negatives (correctly predicted negative samples), false positives (incorrectly predicted positive samples), and false negatives (incorrectly predicted negative samples), respectively. Then precision quantifies the percentage of samples predicted to be positive that are indeed positive:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.20}$$

In IR tasks, precision is the fraction of relevant documents among the retrieved documents.

Recall measures the percentage of positive samples that have been predicted correctly:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.21}$$

In IR tasks, recall is the fraction of relevant documents that have been retrieved by the model.

The F1-score is defined as the harmonic mean of precision and recall:

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.22}$$

It is a balanced metric that combines precision and recall into a single value. There are two common ways to aggregate these scores over a test set: One can either compute precision, recall, and F1-scores for each sample and then average them to obtain a *macro-averaged* score. Alternatively, one can sum the TP, FP, and FN across all samples and then compute precision and recall, which is called *micro-averaging*.

Accuracy

Accuracy measures the percentage of correct predictions over all predictions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.23}$$

It is commonly used in classification tasks with balanced class distributions, but can be misleading when classes are imbalanced. For example, consider a classifier that always predicts the majority class in a binary classification task with a 90% majority class. This classifier would achieve an accuracy of 90%, but would not be useful in practice.

BLEU

For sequence prediction tasks, the BLEU metric is commonly used to evaluate the quality of the generated sequences. It was introduced by Papineni et al. (2002) and is designed to measure the overlap between a model's output and multiple reference sequences. It is commonly used to evaluate machine translation systems (Vaswani et al. 2017). BLEU calculates the precision of n-grams, denoted with B_1, B_2, B_3, B_4 for n-gram sizes 1 to 4, found in the predicted sequence against those in the reference sequences, regardless of their position.

$$B_n = p_n = \frac{\sum_{C \in \mathbb{C}} \sum_{\text{n-gram} \in C} \text{count}_{clip}(\text{n-gram})}{\sum_{C' \in \mathbb{C}} \sum_{\text{n-gram}' \in C'} \text{count}(\text{n-gram}')} \tag{2.24}$$

Additionally, a brevity penalty is applied to penalize shorter predictions. An overall BLEU score is computed as the geometric mean of the n-gram precision scores, modified by the brevity penalty (BP):

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{4} \frac{1}{4} \log B_n\right) \tag{2.25}$$

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-\frac{r}{c}} & \text{otherwise} \end{cases} \tag{2.26}$$

In this formula, c and r represent the lengths of the candidate and the best matching reference sequences, respectively. Here, $\mathbb{C}$ denotes the set of candidate sequences. Most importantly, $\text{count}_{clip}(\text{n-gram})$ clips the count of the n-gram to its maximum count in the reference sequences. To obtain a final corpus-level BLEU score the micro average is computed by summing the nominators and denominators of all prediction and reference(s) pairs.

Information Retrieval Metrics

Information Retrieval (IR) metrics are used to evaluate the performance of IR systems, which retrieve documents based on a user's query or a given document. In IR a large corpus of documents is indexed. To evaluate the quality of the retrieval, a set of queries exists $\mathbb{Q}$, each query q has a set of relevant documents $\mathbb{D}_q$. An IR system retrieves a ranked list of documents for each query, and this section denotes the i-th retrieved document for query q as $d_q^{(i)}$. The result is a hit if the returned document is relevant to the query, i.e., $d_q^{(i)} \in \mathbb{D}_q$. The following metrics are used to evaluate the quality of the ranking of the retrieved documents.

Precision at k Simply using precision or recall for IR tasks can be misleading, since they do not account for the order of the retrieved documents. Hence, in IR tasks, often only the top k retrieved documents are considered, such as the top 10 or top 100, as these are the search results a user would typically inspect. For a single query $q \in \mathbb{Q}$, precision at k (Prec@k) measures the proportion of relevant targets among the top k retrieved results.

$$\text{Prec@}k(q) = \frac{1}{k} \sum_{i=1}^{k} \mathbf{1}_{d_q^{(i)} \in \mathbb{D}_q} \tag{2.27}$$

The recall at k (Recall@k) is defined analogously, but measures the proportion of relevant targets that are retrieved in the top k results.

$$\text{Recall@}k(q) = \frac{1}{|\mathbb{D}_q|} \sum_{i=1}^{k} \mathbf{1}_{d_q^{(i)} \in \mathbb{D}_q} \tag{2.28}$$

Note that both metrics do not account for the order of the retrieved results within the top k results. To obtain an overall score for a test set, Prec@k and Recall@k are averaged over all queries in the dataset.

Average Precision Average Precision (AP) is defined as the mean of the Prec@k scores at each relevant target position and rewards relevant targets earlier in the result list. For a single query q, the AP is calculated as:

$$\text{AP}(q) = \frac{1}{|\mathbb{D}_q|} \sum_{k=1}^{|\mathbb{D}_q|} \text{Prec@}k(q) \cdot \mathbf{1}_{d_q^{(k)} \in \mathbb{D}_q} \tag{2.29}$$

Mean Average Precision The most common evaluation metric among the text retrieval community is Mean Average Precision (MAP) (Manning et al. 2008, p. 159). The MAP is the mean of the AP scores across all queries, and provides a single score that measures the quality across all recall levels.

$$\text{MAP} = \frac{1}{|\mathbb{Q}|} \sum_{q \in \mathbb{Q}} \text{AP}(q) \tag{2.30}$$

Mean Reciprocal Rank Often a user considers only the first relevant document, and the Mean Reciprocal Rank (MRR) metric reflects this by measuring the reciprocal of

the rank of the first relevant document in the list of retrieved documents.

$$\text{MRR} = \frac{1}{|\mathbb{Q}|} \sum_{q \in \mathbb{Q}} \frac{1}{\text{rank}_q} \tag{2.31}$$

where rank_q is the position of the first relevant document in the list of retrieved documents for query q. If none of the retrieved documents are relevant to the query, the reciprocal rank is zero. The MRR is used instead of using the mean rank, since the mean rank can be skewed by outliers (e.g., a single query with a very high rank).

Normalized Discounted Cumulative Gain The aforementioned all operate on binary relevance scores, where a document is either relevant or not. Some IR evaluation benchmarks provide relevance scores for the retrieved documents (e.g., 1 to 4, where 4 is the most relevant). The Normalized Discounted Cumulative Gain (nDCG) metric is designed to take this degree of relevance of a result into account (Manning et al. 2008, p. 163). For a query q with a result list of length k, nDCG is calculated as:

$$\text{DCG@}k(q) = \sum_{i=1}^{k} \frac{2^{\text{Rel}(q, d_q^{(i)})} - 1}{\log_2(1+i)} \tag{2.32}$$

The nDCG is then computed by normalizing the DCG@k by an ideal ranking IDCG@k, in which the relevant documents sorted by their relevance score are at the top of the result list.

$$\text{nDCG@}k(q) = \frac{\text{DCG@}k(q)}{\text{IDCG@}k(q)} \tag{2.33}$$

where $\text{Rel}(q, d_q^i)$ is the relevance score of the i-th document in the result list for query q, and m is the number of relevant documents for query q. To compute an overall nDCG score for a test set, the nDCG scores are averaged over all queries.

2.3 Transformer Model

The transformer architecture as introduced by Vaswani et al. (2017) has been designed for sequence-to-sequence generation tasks (see Section 2.2.2). The architecture as proposed by Vaswani et al. (2017) is used in Chapter 3, while the remaining chapters train models that use an architecture based on the T5 model from Raffel et al. (2020), which contains some minor architectural improvements over the original transformer model, that will be detailed when relevant. This section does not cover all details of the transformer architecture, but focuses on the components and concepts that find application in this thesis. For a more detailed description of to the transformer architecture, the reader is referred to the original paper (Vaswani et al. 2017).

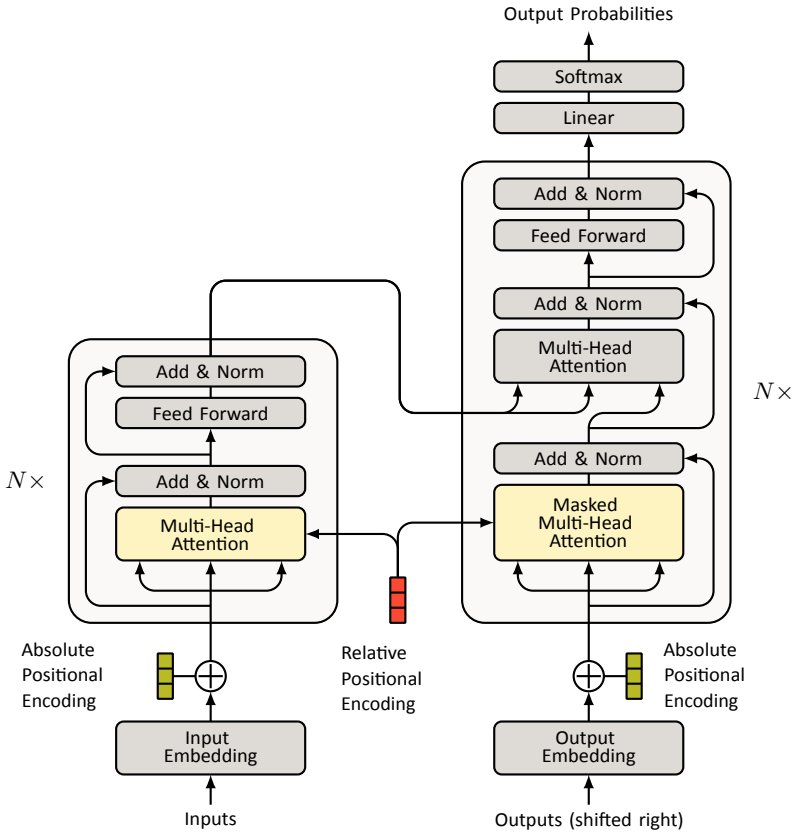


Figure 2.4: Visualization of the transformer architecture, that shows two commonly used techniques to integrate positional information into the model: either absolute or relative positional encodings are used to encode positional information in the transformer. Absolute positional encodings (shown in green) are added to the token-input embeddings to encode positions in the input sequence. Meanwhile, relative positional encodings (red) are integrated into the self-attention mechanism (light yellow). *Illustration adapted from Vaswani et al. (2017).*

First, the transformer architecture is a sequence-to-sequence model, which consists of a *transformer encoder* and a *transformer decoder* (see Figure 2.4). The transformer encoder maps the input vector which consists of tokens indices $\boldsymbol{x} = (x^{(1)}, \ldots, x^{(n)})$ (compare Section 2.1.1), to a series of hidden states or vectors $\boldsymbol{z} = (\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(n)})$, where n is the number of tokens, and each $\boldsymbol{z}^{(i)} \in \mathbb{R}^d$ is the d-dimensional hidden state representing token i. The decoder uses *encoder-decoder attention* to attend to $\boldsymbol{z}$ to generate an output token sequence $\hat{\boldsymbol{y}} = (\hat{y}^{(1)}, \ldots, \hat{y}^{(m)})$ in an autoregressive manner. Architecturally, as visualized in Figure 2.4, both the encoder and decoder consist of multiple stacked layers, each working on embeddings of dimension d. The output of each layer is input to the next one. The output $\boldsymbol{z}$ of the last layer of the encoder is used as additional input to the decoder. The last layers output of the decoder is fed into a prediction head to predict $\hat{\boldsymbol{y}}$, consisting of a linear layer, that maps the hidden state to logits of the output vocabulary size, followed by a softmax layer (compare Section 2.2.2).

Initially, the input tokens are embedded using a learned embedding matrix $\boldsymbol{E}_{\text{in}} \in \mathbb{R}^{|\mathbb{V}| \times d}$, turning $\boldsymbol{x}$ into a sequence of embeddings $(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)})$. The prediction head needs to perform the inverse operation, mapping the hidden states back to the vocabulary space, i.e., requires a weight matrix $\boldsymbol{E}_{\text{out}} \in \mathbb{R}^{d \times |\mathbb{V}|}$. To this end, one has the option to use the same embedding matrix $\boldsymbol{E}_{\text{out}} = \boldsymbol{E}_{\text{in}}^{\top}$ as the weight in the prediction head (this is called tied embeddings), or to use a separate matrix. Note that these matrices contain a considerable amount of the model's parameters.

Each layer contains a multi-head self-attention and a feed-forward sublayer. A residual connection (He et al. 2016) surrounding each sublayer allows the model to propagate information through the layers and is followed by layer normalization (Ba et al. 2016). Self-attention computes new representations for each token by attending to all other tokens in the input sequence. The attention mechanism is the key component of the transformer architecture and is described in detail in the next section. The feed-forward layer, a fully connected network with an activation function[4], operates on each position separately and identically.

The decoder differs from the encoder in two ways. First, its self-attention sublayer is masked to prevent attending to future positions during training. Second, it includes a third sublayer between self-attention and feed-forward, which performs encoder-decoder attention, and allows the decoder to attend to the encoder's output.

2.3.1 Multi-Head Attention

Multi-head attention is the essential component of the transformer architecture (Vaswani et al. 2017). Its purpose is to enable the model to focus on relevant parts of the input sequence while computing a representation for a specific token. The attention mechanism is applied in two distinct forms in the transformer's encoder and decoder: First, self-attention not only allows every token in $\boldsymbol{x}$ to attend to every other token in $\boldsymbol{x}$, but it also allows the decoder to attend to previously generated tokens $\boldsymbol{y}^{(<i)}$. Second, encoder-decoder attention allows the decoder to focus on different parts of the input sequence. Attention effectively computes a weighted sum of linearly transformed *value* embeddings, with the weights derived from a compatibility score between (linearly transformed) *query* and *key* embeddings. The *queries* are the items of interest (Jurafsky and Martin 2009, p. 217), while the *keys* are tokens against which the queries are compared, resulting in the so-called attention weights. The *values* are then weighted and summed based on the attention weights. In self-attention, the query, key, and value embeddings all originate from the same input sequence (either $\boldsymbol{x}$ or $\boldsymbol{y}$), while in encoder-decoder attention, the query

[4] Vaswani et al. (2017) use the Rectified Linear Unit (ReLU) activation function, while the newer T5 model use Gaussian Error Linear Unit (GELU) (Raffel et al. 2020).

embeddings come from the preceding decoder layer, and the key and value embeddings come from the encoder output.

In practice, the attention mechanism is implemented as a multi-head attention mechanism, where the aforementioned attention computation is performed by h attention heads in parallel, each mapping the input sequence to a different d_h-dimensional vector space. After computation their outputs are concatenated and linearly transformed back to dimensionality d, to produce the final attention layer output sequence (see Vaswani et al. (2017) for details). Formally, a self-attention head has parameters $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V \in \mathbb{R}^{d \times d_h}$, that maps the input sequence of embeddings $\boldsymbol{x}=(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)})$ with $\boldsymbol{x}^{(i)} \in \mathbb{R}^d$ to head-specific vector spaces[5]. This sequence can be the output of the previous transformer layer or the input embeddings in the first layer. The self-attention head calculates the attention weight α_{ij} for each token pair i and j, and produces a head-specific output sequence of embeddings $\boldsymbol{z}_h = (\boldsymbol{z}_h^{(1)}, \ldots, \boldsymbol{z}_h^{(n)})$ with $\boldsymbol{z}_h^{(i)} \in \mathbb{R}^{d_h}$ as a weighted sum (compare Shaw et al. (2018) for this section).

$$\boldsymbol{z}_h^{(i)} = \sum_{j=1}^{n} \alpha_{ij} \boldsymbol{x}^{(j)} \boldsymbol{W}^V \tag{2.34}$$

The attention weight α_{ij} is determined by a softmax over the compatibility scores e_{ij}.

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} \tag{2.35}$$

The compatibility score e_{ij} is computed using a scaled dot product of the linearly transformed query and key embeddings.

$$e_{ij} = \frac{\left(\boldsymbol{x}^{(i)} \boldsymbol{W}^Q\right) \left(\boldsymbol{x}^{(j)} \boldsymbol{W}^K\right)^\top}{\sqrt{d_h}} \tag{2.36}$$

The final output of the attention layer is the concatenation of the outputs of all attention heads, linearly transformed to dimensionality d, which is omitted here for brevity.

Essentially, the attention weights α_{ij} indicate how much each token j in the reference sequence contributes to the interpretation of token i. Tokens with higher attention weights have a stronger influence when computing the output embedding $\boldsymbol{z}_h^{(i)}$. The transformer can use this mechanism to contextualize each token by incorporating information from the entire sequence. This allows capturing long-range dependencies much better than traditional Recurrent Neural Network (RNN) architectures.

[5] For encoder-decoder attention, different input sequences are used as outlined above, but this is omitted here for brevity, since the work in Chapter 3 only modifies self-attention.

2.3.2 Positional Embeddings

The transformer architecture as described so far lacks a positional bias for recognizing the sequential order of tokens, because of the absence of recurrence or convolutional operations. Figure 2.4 shows two ways to overcome this problem: One can use either *absolute positional embeddings* (green) which are added to the input embeddings $(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)})$ (Vaswani et al. 2017) or *relative positional embeddings* (red) (Shaw et al. 2018; Raffel et al. 2020) which are integrated into the self-attention mechanism. Absolute positional embeddings encode fixed positions of tokens in the input sequence, while relative positional embeddings encode the relative distance between tokens in the input sequence. In Chapter 3, this thesis proposes a novel way to encode trees with relative positional embeddings.

Absolute Positional Embeddings

Absolute positional embeddings use fixed vectors to represent token positions. They ensure each token's position in a sequence is uniquely identifiable. These embeddings can be either predetermined (e.g., sinusoidal) or learned during training. This positional vector $\boldsymbol{a}^{(pos)} \in \mathbb{R}^d$ is added to the input embedding $\boldsymbol{x}^{(pos)}$ of the token at the corresponding position pos. In order for this approach to be effective, it has to be compatible with the input embedding, and thus the positional vector shares the same dimensionality d as the input embedding of the model. Vaswani et al. (2017) define the sinusoidal positional embedding for a given position pos and dimension i as:

$$a_{2i}^{(pos)} = \sin\left(\frac{pos}{10000^{2i/d}}\right) \tag{2.37}$$

$$a_{2i+1}^{(pos)} = \cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{2.38}$$

An interesting property of sinusoidal embeddings is that they allow the model to attend to future positions, because $\boldsymbol{a}^{(pos+k)}$ is expressible as a linear function of $\boldsymbol{a}^{(pos)}$, which allows learning temporal relations in the attention mechanism.

Relative Positional Embeddings

Relative positional embeddings, on the other hand, determine the positional embedding between token pairs based on their relative positions in the input sequence (Shaw et al. 2018). The authors define the positional relationship between tokens i and j as the number of tokens separating them, as shown in Figure 2.5, limited to a maximum k.

$$\text{dist}(i, j) = \min(\text{abs}(j - i), k) \tag{2.39}$$

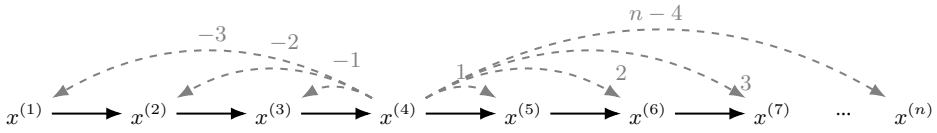


Figure 2.5: Visualization of some relative distances in a sequence of tokens.

A tensor $\boldsymbol{R} \in \mathbb{R}^{d_h \times (k+1) \times 2}$ serves as an embedding table for storing the relative positional embeddings. The embedding table is indexed by the order and the positional difference between tokens i and j to retrieve the corresponding embedding $\boldsymbol{R}_{:,dist(i,j),\mathbf{1}_{i<j}} \in \mathbb{R}^{d_h}$ for each positional pattern. The indicator function $\mathbf{1}_{i<j}$ is 1 if i is to the left of j and 0 otherwise. This enables the transformer to recognize when the order of tokens i and j is interchanged. To enhance the readability, the relative positional embedding for tokens i and j is denoted as $\boldsymbol{r}_{ij} \coloneqq \boldsymbol{R}_{:,dist(i,j),\mathbf{1}_{i<j}}$. The positional embedding is shared between attention heads, but not across layers.

Note that two distinct positional embeddings $\boldsymbol{r}_{ij}^V, \boldsymbol{r}_{ij}^K$ can be integrated at different stages of the attention mechanism: (1) during the weighted sum computation, influencing the values, and (2) during compatibility calculation, modifying the keys. In the first case, the attention weighted sum from Equation (2.34) is modified to include the relative positional embedding $\boldsymbol{r}_{ij}^V$ between tokens i and j:

$$\boldsymbol{z}_h^{(i)} = \sum_{j=1}^{n} \alpha_{ij} (\boldsymbol{x}^{(j)} \boldsymbol{W}^V + \boldsymbol{r}_{ij}^V) \tag{2.40}$$

In the second case, the compatibility score e_{ij} from Equation (2.36) is augmented with relative positional embedding $\boldsymbol{r}_{ij}^K$ when comparing keys and queries:

$$e_{ij} = \frac{\boldsymbol{x}^{(i)} \boldsymbol{W}^Q \left(\boldsymbol{x}^{(j)} \boldsymbol{W}^K + \boldsymbol{r}_{ij}^K \right)^\top}{\sqrt{d_h}} \tag{2.41}$$

In practice, Shaw et al. (2018) found using (1) and (2) performs equally well as using only (2)[6]. The authors also found that maximum distances $k \geq 2$ does not significantly improve performance. They argue that this could be because, even though a single transformer layer cannot directly use positional information beyond this upper bound, it can be propagated to subsequent layers, which can infer it.

More recent works, such as the T5 architecture used in Chapters 4 to 7, use a bucketed approach instead of the simple clipping, e.g., this token is next to you, these are near, these are far away, and so on. Raffel et al. (2020) categorize the distances into 32 buckets that

[6]Specifically, approaches (1) & (2) and only (2) achieve 25.8 BLEU, while using only approach (1) achieves 25.3 BLEU on the WMT 2014 English-German translation task.

represent logarithmically increasing distances, starting from 1 until a maximum distance of 128. Each bucket has two learnable embeddings $\boldsymbol{r}_{ij}^K, \boldsymbol{r}_{ij}^V$. The transformer can refine the bucketed positions in subsequent layers, as with clipping. Furthermore, the T5 architecture does not use vector embeddings ($\boldsymbol{r}_{ij} \in \mathbb{R}^{d_h}$), but instead add scalars element-wise, i.e., $r_{ij} \in \mathbb{R}$, which are shared across all heads and layers.

2.4 Self-Supervised Learning and Language Models

Supervised learning algorithms rely on a training set of input examples $\boldsymbol{x}$ and output examples $\boldsymbol{y}$ (Goodfellow et al. 2016, p. 136). During training, a machine learning model learns to associate an input with its corresponding output (as for the tasks described in Section 2.2.2). In contrast, *unsupervised learning* "refers to most attempts to extract information from a distribution that do not require human labor to annotate examples" (Goodfellow et al. 2016, p. 142). *Self-supervised learning* is a mixture of unsupervised and supervised learning, as it creates supervised training data without human intervention.

Annotating large datasets for supervised learning is cost-intensive, so that *self-supervised pretraining strategies* have gained popularity. These strategies aim to build an initial understanding of the underlying data using an auxiliary task, such as *language modeling*. After pretraining, the internal knowledge can be applied to a downstream task, which is also called *transfer learning* (Goodfellow et al. 2016, p. 536). Transfer learning assumes the variations and patterns learned during pretraining to be also useful for the downstream task. This is a powerful concept, as it allows models to be trained on large amounts of unlabeled data, and then used for a variety of tasks, even if only a small amount of labeled data is available for the target task—sometimes even without any labeled data at all, which is called *zero-shot*. Self-supervised pretraining tasks typically operate on unimodal data, such as large collections of code or text, and do not require human annotation. Instead, they bootstrap input/output pairs for supervised learning with auxiliary tasks.

Language Models (LMs) are a popular self-supervised learning strategy for NLP tasks. Generally speaking, an LM defines a probability distribution over sequences of tokens (Goodfellow et al. 2016, p. 461), and is trained to predict parts of a sequence based on the context. How context and the part to predict are defined varies between different models and learning objectives. In the last 10 years, LMs have been subject to intensive study, leading to three distinct generations of language models. In the following sections, these different generations of LMs and their associated self-supervised learning strategies are discussed.

2.4.1 Word Embeddings

It has long been known that words that occur in similar contexts have similar meanings (Firth 1957). Word embeddings adopt this idea, where terms or documents are represented

as dense vectors $\boldsymbol{v} \in \mathbb{R}^d$ of smaller dimensionality in a latent space, so that similar terms are close to each other and relations between terms can be captured (Mikolov et al. 2013c). The Continuous-Bag-Of-Words (CBOW) approach to learning word embeddings proposed by Mikolov et al. (2013a) is trained using a self-supervised cloze task objective on a large corpus of text. For each word $w^{(t)}$ in a text corpus of T tokens, the model is trained by minimizing the negative log-likelihood of the central word in a given context window of l words:

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{T}\sum_{t=1}^{T} \log p(w^{(t)} \mid w^{(t-\lfloor l/2 \rfloor)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+\lfloor l/2 \rfloor)}) \tag{2.42}$$

The CBOW model stores two matrices $\boldsymbol{E}_{\text{in}} \in \mathbb{R}^{|\mathbb{V}| \times d}$ and $\boldsymbol{E}_{\text{out}} \in \mathbb{R}^{d \times |\mathbb{V}|}$, where $\boldsymbol{E}_{\text{in}}$ is the input and $\boldsymbol{E}_{\text{out}}$ is the output embedding matrix. The input embedding $\boldsymbol{v}^{(t)}$ of the t-th word is obtained by averaging the embeddings of the context words:

$$\boldsymbol{v}^{(t)} = \frac{1}{l-1} \sum_{i=-\lfloor l/2 \rfloor, i \neq 0}^{\lfloor l/2 \rfloor} \boldsymbol{E}_{\text{vocab}(w^{(t+i)}),:} \tag{2.43}$$

where vocab is a mapping from words to indices in the vocabulary, as defined in Equation (2.1). The input embedding is then compared to all output embeddings to obtain a probability distribution over the vocabulary[7], that is used to predict the central word:

$$p(w^{(t)} \mid w^{(t-\lfloor l/2 \rfloor)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+\lfloor l/2 \rfloor)}) = \text{softmax}(\boldsymbol{E}_{\text{out}} \cdot \boldsymbol{v}^{(t)}) \tag{2.44}$$

After training, the word embeddings in $\boldsymbol{E}_{\text{in}}$ can be used for nearest neighbor searches to obtain similar words, e.g., retrieving identifier `db` for identifier `database`. Interestingly, the embeddings capture notions of semantic similarity, and certain relations manifest as vector offsets (Mikolov et al. 2013c).

One key problem with word embeddings is that the embeddings are static and do not consider the context in which the word is used. Hence, static word embeddings can not distinguish between words that have different meanings, such as homonyms. For example, the words *bow* (bend forward) and *bow* (a weapon) will have the same embedding.

Subword Embeddings

Another disadvantage of word embeddings—as described above—is that they cannot represent words that are not in the vocabulary. This is especially problematic for source

[7] Note that for large vocabularies, computing the softmax over the vocabulary is computationally expensive or infeasible. Mikolov et al. (2013a) propose to use a hierarchical softmax implementation or negative sampling to approximate the softmax. This is not further discussed here.

code identifiers, which are often domain-specific and contain compound words, such as `databaseConnection`. For example, during training time when building the vocabulary, one may only encounter the identifiers `database` and `connection`, but not the compound word. During inference no embedding is available for the compound word, and one would need to perform manual post-processing and use the average of the embeddings of the individual words.

Subword embeddings, such as the FastText embeddings proposed by Bojanowski et al. (2017), aim to solve this problem by building the embedding for a word from its subword information automatically. Specifically, the authors propose to encode multiple character n-grams of the word and compute Equation (2.44) for each n-gram individually. The embeddings of the subwords are summed to form the embedding of the word. This has the advantage that the amount of possible subwords is much smaller than the amount of words, and the model can create embeddings for unseen words at inference time by combining the embeddings of the subwords.

2.4.2 Contextualized Language Models

Compared to static word embeddings, *contextualized language models* generate word representations that depend on the context in which the word appears. For example, when building a representation for the word *bow* in the sentence "*He took a bow after the performance*", the model can use the context to distinguish that *bow* has a different meaning and thus should get a different representation than in the sentence "*He used a bow to shoot the arrow*". While some language models are built with n-grams (Liu et al. 2024) or with Long-Short-Term-Memory Networks (LSTMs) (Peters et al. 2018), the main driver and most successful architecture for contextualized language models has been the transformer architecture (see Section 2.3). The transformer architecture serves as the foundation for many widely used LMs in NLP and IR, such as Bert (Devlin et al. 2019), GPT (Radford et al. 2018), and T5 (Raffel et al. 2020). Vaswani et al. (2017) proposed an encoder-decoder transformer architecture, as detailed in Section 2.3, but the individual components can also be used separately. For example, the Bert model uses only the transformer encoder, while the GPT model uses only the decoder part. LMs from this generation are typically pretrained on large text corpora using self-supervised learning objectives, and then *fine-tuned* on specific downstream tasks, with much smaller labeled datasets.

Masked Language Modeling

One popular self-supervised learning objective is *Masked Language Modeling (MLM)*, as used in the Bert, CodeBert, and RoBERTa models (Feng et al. 2020; Liu et al. 2019). It is designed for the transformer encoder, and allows learning bidirectional contextual representations for the input sequence (i.e., modeling $p(\boldsymbol{x})$), since the self-

attention mechanism in the encoder considers both left and right contexts. This makes MLM powerful for applications that need to build a representation for a complete input sequence, such as text classification or IR.

The model is trained to predict masked tokens in the input sequence. Formally, a corrupted input sequence $\tilde{\boldsymbol{x}}$ is created from $\boldsymbol{x} = (x^{(1)}, x^{(2)}, \ldots, x^{(n)})$ by modifying a set of positions $\mathbb{M} \subseteq \{1, 2, \ldots, n\}$. In 80% of the cases, the tokens at these positions are replaced with a special mask token $x^{(mask)}$ (see Section 2.1.1). In 10% of the cases, the tokens are replaced with a random token from the vocabulary, and in 10% of the cases, the tokens are left unchanged (Jurafsky and Martin 2009, p. 248). The transformer encoder maps $\tilde{\boldsymbol{x}}$ to a sequence of hidden states $\boldsymbol{z}$, as detailed in Section 2.3. The model is then trained to predict the original tokens at the masked positions:

$$\mathcal{L}_{\text{MLM}}(\boldsymbol{\theta}) = -\frac{1}{|\mathbb{M}|} \sum_{i \in \mathbb{M}} \log p_{\text{model}}(x^{(i)} \mid \boldsymbol{z}^{(i)}) \tag{2.45}$$

The first token of the sequence is always a special classification token $x^{(cls)}$, which is used to generate a representation for the complete sequence. This representation is used for downstream tasks, such as text classification or IR.

Autoregressive Language Modeling

Another self-supervised learning objective is *autoregressive language modeling*, as used in the GPT, InCoder, and StarCoder models (Radford et al. 2018; Fried et al. 2023; Li et al. 2023). In this setting, the model predicts each token in the sequence given the previous tokens. This model uses the transformer decoder, which is designed to generate sequences autoregressively, hence, the input is denoted as $\boldsymbol{y} = (y^{(1)}, \ldots, y^{(n)})$. The model is trained to maximize the likelihood of the sequence (i.e., modeling $p(\boldsymbol{y})$) by predicting the next token given the previous tokens:

$$\mathcal{L}_{\text{ALM}}(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \log p_{\text{model}}(y^{(i)} \mid \boldsymbol{y}^{(<i)}) \tag{2.46}$$

Note that this loss corresponds to Equation (2.16) without the conditioning on the input sequence $\boldsymbol{x}$, as the model generates the output sequence from scratch. An autoregressive language model is typically used for text or code completion tasks (GitHub 2024).

However, the autoregressive nature makes using the available context more difficult. Consider for example code completion, where the cursor marks the position at which the completion should be inserted. The model has to generate the completion based on what comes before in the file, which may miss some important methods that are defined later in the file (e.g., a helper function). The InCoder model addresses this issue by inserting

marker tokens at the position of the cursor, and using another token to indicate the end of the context.

Sequence-to-Sequence Language Modeling

Finally, encoder-decoder models, such as T5, CodeT5, and BART (Raffel et al. 2020; Wang et al. 2021b; Lewis et al. 2020b), extend the above ideas by using a sequence-to-sequence framework, where the model is trained to map a noised input sequence $\boldsymbol{x}$ to an output sequence $\boldsymbol{y}$. Typically, the input sequence is corrupted by deleting, shuffling, or replacing tokens or spans of tokens, and predicting either the original sequence or the missing parts. This setup allows using the regular sequence prediction loss for training, as detailed in Section 2.2.2 and Equation (2.16). Such an encoder-decoder LM for code is developed in Chapter 4, which also provides additional details on the training process, model architecture, and pretraining tasks.

2.4.3 Large Language Models

Recently, LLMs—mostly from the decoder-only category—have been scaled up to multiple billions of parameters, resulting in the development of Large Language Models (LLMs), such as GPT-3, GPT-4, and Llama3 (Brown et al. 2020; OpenAI 2023; Dubey et al. 2024), has shifted the focus from fine-tuning to *prompting*. LLMs are scaled-up autoregressive transformer decoder models, and their amount of parameters ranges from 7 billion (smaller Llama variants), over 175 billion in GPT-3, to an estimated 400 billion in GPT-4. These models are capable of performing a wide range of tasks through *in-context learning*, where the model learns to perform a task based on examples provided in the input prompt, or is able to do so without any examples at all. LLMs are trained with Equation (2.46) on massive amounts of data and compute. After pretraining, they are capable of generating coherent text, but are mainly still autocompletion models predicting the next token, which is not always the output desired by the user. For code completion tasks, however, this setting is particularly useful, as the model simply continues writing the program, conditioned on the input code. Hence, tools such as CodeX (Chen et al. 2021), Tabnine (Tabnine 2024), and GitHub Copilot (GitHub 2024) have been developed based on these models, and have shown impressive capabilities in code generation tasks. Moreover, to improve the ability of LLMs to follow user instructions, *instruction tuning* (Ouyang et al. 2022) has been found to be effective. Models like ChatGPT (OpenAI 2024a) fine-tune the LLM once using reinforcement learning with human feedback to follow user instructions. After this training step, the instruction-tuned LLM can be applied to a wide range of tasks, all based on the prompt given to the model. This marks a shift in the development of LMs, moving away from fine-tuning on downstream tasks and towards LLMs as "general problem solvers" with prompt-based learning.

While LLMs have recently demonstrated impressive capabilities, they have been developed concurrently with (or after) the work presented in this thesis, and are not the focus of this thesis. Nevertheless, the advancements in LLMs present exciting opportunities for future research, particularly in enhancing the models and techniques discussed in this thesis. For instance, in Chapter 5, the code search embeddings are compared with those from OpenAI's LLMs. Additionally, the conclusion in Chapter 8 explores how LLMs can be integrated with the proposed approaches and tools to further improve performance in code-related tasks.

2.5 Information Retrieval

This section has been largely influenced by the excellent books *Introduction to Information Retrieval* (Manning et al. 2008) and *An Introduction to Neural Information Retrieval* (Mitra and Craswell 2018). It provides a brief overview of the development of Information Retrieval (IR) techniques and introduces the ones used in this work. It starts with traditional keyword-based retrieval, followed by distributional semantics and neural IR.

IR is the process of finding relevant documents or passages in a collection of unstructured documents that satisfy a user's information need (Manning et al. 2008, p. 1). In this thesis, documents and queries are sequences of tokens from a vocabulary $\mathbb{V}$, produced by a tokenization process (see Section 2.1.1). The goal is to rank the documents based on their relevance to the query.

2.5.1 Keyword-Based Information Retrieval

Traditional IR systems employed probabilistic keyword-based retrieval techniques and have been used in many code search systems, e.g., (Sindhgatta 2006; Grechanik et al. 2007; Grechanik and Poshyvanyk 2008; Chatterjee et al. 2009; Bajracharya et al. 2010; Lv et al. 2015). Sparse *vector space models* that encode a piece of text as a high-dimensional vector $\boldsymbol{v} \in \mathbb{R}^{|\mathbb{V}|}$ have been used for decades (Manning et al. 2008, p. 120). Based on these vectors, the similarity between two texts can be calculated using cosine similarity as in Equation (2.18). Ranking is then performed by comparing the similarity of the query to all documents or passages in the corpus. The vectors $\boldsymbol{v}$ are called *bag-of-words* vectors, since they do not consider the order of the words in the text. Several weighting schemes for $\boldsymbol{v}$'s entries have been proposed to improve retrieval quality compared to a one-hot encoding, starting with term frequency, TF-IDF (Term Frequency-Inverse Document Frequency) (Sparck Jones 1972), and later BM25 (Robertson et al. 1994; Robertson and Zaragoza 2009). The following sections provide a brief overview of these weighting schemes.

TF-IDF

Let $\text{tf}(D, t)$ denote the frequency of token $t \in \mathbb{V}$ in a tokenized document D, and $\text{df}(t)$ the number of documents in the corpus that contain t. The Inverse Document Frequency (IDF) is subsequently defined as $\text{idf}(t) = \log_{10} \frac{N}{\text{df}(t)}$, where N is the total number of documents in the corpus (Jurafsky and Martin 2009, p. 296). Intuitively, the IDF gives a high score to "informative" terms appearing in few documents, and a low score to terms appearing in many documents and thus having a low entropy. Terms appearing in every document, such as stopwords, have an IDF of zero.

Then the TF-IDF weight of term t in document D is calculated as follows:

$$\text{tf-idf}(D, t) = \text{tf}(D, t) \cdot \text{idf}(t) \tag{2.47}$$

One can think of the TF-IDF weight as a measure of how important a term is in a document relative to the corpus. This allows to define the aforementioned sparse vector $\boldsymbol{v}$ for a document D (or a query) so that $v_i = \text{tf-idf}(D, \mathbb{V}^{(i)})$ for all terms in the vocabulary $\mathbb{V}$ (see Section 2.1.1).

BM25

BM25 proposed by Robertson et al. (1994) is an extension to TF-IDF, which can be considered the industry standard for enterprise IR, and is used by many practical search engines such as ElasticSearch. Hence, it is used as a baseline in Chapter 5. Jurafsky and Martin (2009, p.298) define the BM25 ranking function as follows for a query $Q = (q^{(1)}, q^{(2)}, \ldots, q^{(m)})$ and a document D:

$$\text{BM25}(D, Q) = \sum_{i=1}^{m} \text{idf}(q^{(i)}) \cdot \frac{\text{tf}(D, q^{(i)})}{k \left(1 - b + b(\frac{\text{length}(D)}{\text{length}(D_{\text{avg}})}) + \text{tf}(D, q^{(i)})\right)} \tag{2.48}$$

where k and b are free parameters, and $\text{length}(D_{avg})$ are the average document length in the corpus. The higher the score, the more relevant the document is to the query. One can see that BM25 is similar to TF-IDF, but with an additional term in the denominator that introduces term frequency saturation and document length normalization (Jurafsky and Martin 2009, p. 298). The parameters k and b adjust how strongly term frequency influences relevance and account for document length, which is more flexible compared to TF-IDF.

2.5.2 *Distributional Semantics*

The high-dimensional vector space models have the drawback that the representation does not convey that *apple* and *pear* are more similar than *apple* and *communism*. "But when items have distributed or feature based representations, then the similarity between two

items is determined based on the similarity between their features" (Mitra and Craswell 2018, p. 29). This is the foundation of modern neural-based retrieval models, which represent tokens by dense *embeddings*. Earlier approaches factorized the term-feature matrix to obtain embeddings, such as LSA (Deerwester et al. 1990) or PLSA (Hofmann 1999). Nowadays, embeddings are learned with neural networks by setting up a feature prediction task (Mitra and Craswell 2018, p. 38). This is also referred to as *learning to rank*. For example, by using the contrastive learning objective detailed in Section 2.2.2 to train a model to assign similar embeddings to a related query and document.

Supervised Neural Information Retrieval

Supervised approaches use human relevance assessments, i.e., query-result pairs, to optimize a neural model for a ranking task. Earlier versions of these models were trained end-to-end with a loss function that directly optimizes the ranking quality (Huang et al. 2013; Gillick et al. 2018). Due to their simplicity and fast retrieval, the aforementioned models and most others use bi-encoders, as described in Section 2.2.2. These approaches were boosted by the development of LMs (see also Section 4.2), such as BERT, which have been used as a basis for many IR models (Nogueira et al. 2019; Humeau et al. 2020). However, bi-encoders have been found to be outperformed by supervised cross-encoders that encode the query and the document together (Nogueira et al. 2019; Humeau et al. 2020). Allowing the model to combine every word in the query with every word in the document is surely more expressive than the bi-encoder approach, but also requires encoding the query together with every document in the corpus with the model at inference time. This is computationally much more expensive, if not infeasible, for large corpora. To this end, Nogueira et al. (2019) proposed a multi-stage ranking approach that combines the best of both worlds by first retrieving a large set of candidates with BM25 and then reranking them with a cross-encoder.

The DPR model (Karpukhin et al. 2020) was one of the first models to successfully use bi-encoders for supervised IR. It has been shown to outperform BM25 on the MS MARCO dataset (Nguyen et al. 2016) by using in-batch negative samples along with a hard-negative sample with high keyword overlap that does not contain the answer (retrieved with BM25). Also, contrastive learning with bi-encoders has been successfully applied to learn sentence similarity with aligned sentence pairs (Reimers and Gurevych 2019). Luan et al. (2021) compare different neural ranking architectures and find that, while cross-encoders outperform bi-encoders (which in turn outperform BM25), hybrid models can combine the strengths of both approaches.

Self-Supervised Learning for Information Retrieval

One of the main challenges in supervised IR is the lack of labeled data, as it is expensive to obtain relevance assessments for query-document pairs, which is why self-supervised

strategies for IR have been proposed. This thesis explores such an approach for an IR task in the context of code search, and develops a self-supervised contrastive learning framework for CCS. This section provides an overview of self-supervised learning for IR and related work.

Distributed representations for words or documents have been developed over the years and have been used in IR to learn semantic similarities between words or documents. From Bengio et al. (2000) to word embedding models such as Word2Vec (Mikolov et al. 2013b) and GloVe (Pennington et al. 2014) to Bert, (contextualized) embeddings have become a standard in NLP. Self-supervised embeddings can be used for IR tasks by aggregating the embeddings of the words in a document or query to obtain a representation. However, while this representation encodes some aspects of similarity, they are not optimized for retrieval. To this end, Lee et al. (2019) train a bi-encoder retriever with an inverse cloze task. Another line of research has explored joint training of retrievers and generators, particularly for open-domain question answering. For instance, REALM (Guu et al. 2020) employs joint training of an encoder LM with a bi-encoder retriever, while Lewis et al. (2020a) train both a retriever and a generator with an unsupervised multilingual multi-document paraphrasing objective. Similarly, Izacard and Grave (2021) propose using cross-attention scores from a sequence-to-sequence model as a training signal for the retriever. Another relevant approach is SimCSE (Gao et al. 2021b), which train a contrastive model by feeding the same sentence twice through the model and assume that dropout adds sufficient noise to the input to learn a good representation. In CCS, the input sequences are different (context and target), but both are encoded independently with dropout by the transformer. This setup is similar to SimCSE but more challenging because it involves paired data.

Part I

Models and Techniques

If you have a good name for a method you don't need to look at the body.

— Fowler (1999, p. 64)

3

Relative Structural Transformers

3.1 Introduction and Motivation

The increasing interest in using machine learning algorithms to semantically understand source code has opened up many possibilities, including identifying code clones (Yu et al. 2019), generating automated summaries (Fan et al. 2018), detecting defects (Zhou et al. 2019), querying databases using natural language (Xu et al. 2017), assigning bugs (Mani et al. 2019), and performing semantic searches of code (Gu et al. 2018). At the time of this research—which took place in 2018 and 2019—the transformer architecture (Vaswani et al. 2017) has been successful in many sequence-to-sequence tasks, such as machine translation and abstractive summarization. However, at the time of this research the transformer has not been widely used for source code understanding. This chapter focuses specifically on evaluating the effectiveness of a transformer-based model for encoding source code. To accurately measure the model's understanding of source code, it is trained end-to-end without pretraining on tasks that encode source code and generate natural language descriptions[1].

This chapter is adapted from **Johannes Villmow**, Adrian Ulges, and Ulrich Schwanecke (2021b). A Structural Transformer with Relative Positions in Trees for Code-to-Sequence Tasks. In *International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021*. IEEE, pp. 1–10, previously published by ©2021 IEEE.

[1] Training a model end-to-end on a dataset has been common in 2018, however, since then Artificial Intelligence (AI) research has shifted to pretraining models on large datasets before fine-tuning them on specific tasks. This is the focus of the next chapter.

An important aspect of understanding source code is capturing its hierarchical structure. Hence, the most successful code models, such as Code2vec (Alon et al. 2019b) and Code2seq (Alon et al. 2019a), rely on abstract representations of syntactic elements, such as AST or data flow graphs (Guo et al. 2021). The authors find that syntactic representations provide a richer representation of the code's semantics to the model than a linear sequence of tokens. Obviously, the model does not need to learn the structure, when it is provided with a syntactic prior. The model can focus on the actual task and—hypothetically—requires less training data to achieve the same performance. In 2019, transformers have become the state-of-the-art for modeling all kinds of tasks in NLP, for example with Bert. However, the transformer architecture is designed to efficiently process sequences of tokens, and fails with structural inputs such as trees, primarily due to its inherent limitation in recognizing hierarchical structures. Overall, self-attention, the transformer's main component, is based on a pairwise comparison between all tokens in the input, whereas "structure" (i.e., the linear order of tokens) is only encoded by adding positional embeddings to the input. The overall research goal of this chapter is to investigate how to encode structural information from trees with transformers to improve their performance on sequence prediciting code understanding tasks.

Integrating trees into transformers can be achieved through three general strategies: (1) aggregation (Nguyen et al. 2020), (2) the use of positional embeddings (Shiv and Quirk 2019), and (3) a structure-aware loss function (a novel contribution of this chapter). Aggregation is proposed by Nguyen et al. (2020), who modify the attention mechanism to hierarchically aggregate node embeddings. However, this approach suffers from high memory consumption and slow training. It requires quite drastic architectural modifications of the attention mechanism, which might not be necessary. The second approach enhances positional embeddings to encode the structure of trees. As mentioned above, positional embeddings are used to encode the order of tokens in the input sequence. As detailed in Section 2.3.2, they come in two variants in regular transformers: absolute (Vaswani et al. 2017) and relative (Shaw et al. 2018). Shiv and Quirk (2019) have extended absolute positional embeddings to trees. They convert trees into binary trees and introduce a static encoding for binary tree structures, similar to sinusoidal positional embeddings for sequences. However, to the best of the author's knowledge, relative positional embeddings have not yet been used to encode tree structures. The third approach, a structure-aware loss function, is introduced in this chapter.

3.1.1 Contributions

This chapter addresses the aforementioned research gap by introducing a novel approach to integrate structural information from trees into transformers. The approach consists of two main contributions: (1) relative positional embeddings for trees and (2) a structure-aware loss function and is visualized in Figure 3.1. The first extends relative positional

embeddings to encode the structural relationship between tree nodes, which adds an inductive bias for tree hierarchies to transformers. We investigate two different relative positional patterns: the path length between nodes and the upwards and downwards steps in the tree that are required to move from one node to another. Additionally, this chapter demonstrates how these relative positional patterns can be computed efficiently during training. The second contribution is a novel loss function, that predicts the LCA of nodes in the tree from the encoder's output embeddings., which forces the model to preserve structural information within its hidden states. This chapter validates the effectiveness of this approach on three sequence-to-sequence tasks: method naming, code summarization, and machine translation.

- The method naming task aims to predict the name of a method based on its body and signature. It can be seen as an extreme abstractive summarization task and is used to evaluate the model's ability to understand code (Alon et al. 2019a).
- The code summarization task predicts a natural language description of a code snippet. This is also an abstractive summarization task—although with longer output sequences than method naming—that is frequently used for evaluating code understanding models (Alon et al. 2019a).
- The machine translation task translates a natural language text from one language to another. This task is used to evaluate the model's ability to process natural language and is a common benchmark for sequence-to-sequence models in the NLP community (Vaswani et al. 2017).

Since this chapter focuses on sequence-to-sequence tasks, the contributions are integrated into an encoder-decoder transformer model, which is called Relative Structural Transformer (Rst). The results show that Rst outperforms the state-of-the-art on the method naming task by 6 percentage points (p.p.) F1-score and achieves competitive results on the other tasks, while improving over a regular token-based transformer baseline on all tasks. In summary, the key contributions of this chapter are:

- The demonstration that code-to-sequence tasks can be approached end-to-end with self-attention based transformers with relative positional embeddings.
- The introduction of new relative positional tree patterns for self-attention with relative positional embeddings that represent movements in the tree structure, exploring two positional patterns: the path length between nodes and explicitly encoding the upwards and downwards steps in the tree.
- A new loss function that predicts the LCA of nodes in trees for training transformers on tree-structured data.
- Outperforming the state-of-the-art on the method naming task by 6 p.p. F1-score.

3.2 Related Work

This chapter intersects the fields of software engineering and NLP (we encode code and generate natural language). Relevant preliminary work in these areas is outlined separately below. These fields are currently an area of intensive research and aim to replace manual feature engineering with fully automated *representation learning*, which is called deep learning (LeCun et al. 2015). The increasing availability of large datasets, combined with advancements in hardware—particularly GPUs—has enabled the training of large AIs models for complex, domain-specific problems. To support the development of these models, powerful libraries such as TensorFlow and PyTorch (Paszke et al. 2019) are widely used in practice.

3.2.1 *Natural Language Processing*

The field of NLP has made significant progress in recent years, driven by increasing data volumes[2] and new representation learning approaches. Early representation learning approaches, such as Word2Vec and GloVe (Mikolov et al. 2013a; Pennington et al. 2014), proposed vector representations (embeddings) for words in a latent space, where semantically similar words are placed close to each other, and syntactic and semantic relationships are captured as translations (Mikolov et al. 2013c). Typically, these static embeddings were used as inputs to more complex, task-specific neural network architectures, including Convolutional Neural Networks (CNNs) (Kim 2014), RNNs with LSTM units (Hochreiter and Schmidhuber 1997), graph convolutional networks (Kipf and Welling 2017), and recursive neural networks (Socher et al. 2013; Tai et al. 2015). However, RNNs and recursive neural networks are slow to train due to their sequential nature. CNN-based models, such as the one by Dauphin et al. (2017), offer better scalability but were initially found to underperform compared to RNNs on NLP tasks, although later architectures improved their performance (Gehring et al. 2017; Dauphin et al. 2017).

These statistical models based on embeddings achieved significant progress over manual feature engineering approaches across a large field of application areas (Goldberg 2016), including document search (Huang et al. 2016), machine translation (Sutskever et al. 2014), question answering (McCann et al. 2018), text summarization (McCann et al. 2018), relation extraction (McCann et al. 2018; Zhang and Wang 2015), and sentiment analysis (Tai et al. 2015). Some models also generalize to new terms outside the vocabulary (Mikolov et al. 2018).

At the time of this research, the focus in NLP has shifted from approaching the aforementioned tasks with static word embeddings and specialized architectures to attention-based

[2] For example, the open source platform GitHub offers access to over 10 million repositories with freely accessible source code, large text corpora are freely available (Wikipedia, News-Corpora), and also collaboratively maintained knowledge graphs such as DBPedia are a useful data source.

transformer models (Vaswani et al. 2017) that learn embeddings and the task simultaneously. As detailed in Section 2.3, transformer models offer a parallelizable attention mechanism that shortens training time and contextualizes embeddings by aggregating information from surrounding words. This architecture has been successful in a wide range of NLP tasks, including machine translation (Vaswani et al. 2017) and question answering (Devlin et al. 2019). Earlier summarization approaches employed pointer networks (Vinyals et al. 2015) to mitigate the OOV problem by copying words directly from the input sequence (See et al. 2017). However, recent attention has shifted towards using BPE (Sennrich et al. 2016), which addresses the OOV problem by splitting words into a fixed set of subword units (see Section 2.1.1). BPE is utilized in nearly all modern transformer architectures and language models, and therefore used for every approach in this thesis. Note that related work on pretrained models, such as Bert (Devlin et al. 2019), CodeBert (Feng et al. 2020), RoBERTa (Liu et al. 2019), and GPT (Radford et al. 2018), is not covered in this chapter, as the focus is on training models end-to-end on specific tasks. Pretrained models are the focus of the next chapter.

Sequence-to-Sequence Tasks

Sequence-to-sequence tasks, such as machine translation and abstractive summarization, involve transforming a source sequence into a target sequence, and are often approached with encoder-decoder model architectures (Sutskever et al. 2014). In particular, abstractive summarization condenses a source sequence into a concise, descriptive target sequence while preserving its semantic meaning (Fan et al. 2018). This task has been approached using various architectures, including RNNs with optional attention mechanisms (Bahdanau et al. 2015), convolutional networks (Gehring et al. 2017; Fan et al. 2018), and attention-based transformers (Vaswani et al. 2017). However, this chapter follows the aforementioned most recent trends in NLP and uses an encoder-decoder transformer architecture for the experiments.

3.2.2 Machine Learning in Software Engineering

Modeling the semantics of source code using representation learning approaches—such as predicting precise natural language summaries or missing identifiers—has emerged as a research topic in recent years (Allamanis et al. 2018), with applications in clone detection (Baxter et al. 1998), code summarization (Alon et al. 2019a), natural language database querying (Xu et al. 2017), bug triage (Mani et al. 2019), and semantic code retrieval (Gu et al. 2018). Some approaches develop probabilistic models for source code (Bielik et al. 2016; Raychev et al. 2016), which are then used for the deobfuscation of packed Android applications (Bichsel et al. 2016), automated renaming of variables with more meaningful identifiers within JavaScript applications (Raychev et al. 2019), and code completion (Raychev et al. 2014). Similarly, Allamanis et al. (2015) predict identifiers

in source code using a log-bilinear context model. Allamanis et al. (2016) introduce the ConvAttention model that uses a convolutional attention network over the tokens in a method to predict sub-tokens in method names. Recently, the Bert model has been adapted to the source code domain; for example, Feng et al. (2020) train CodeBert on pairs of natural language and methods.

Structural Properties of Source Code

Most approaches treat source code as a token sequence and do not exploit additional structural information provided by existing syntax parsers (Allamanis et al. 2014; Hellendoorn and Devanbu 2017). However, source code is inherently structural—unlike natural language—due to the design of programming languages. A programming language is built on a specific context-free grammar that defines keywords, identifiers, and structure. This grammar is used during syntax analysis to verify the correctness of a program and to represent the program as a tree structure. Several successful approaches in the source code domain utilize that structural information. Tai et al. (2015) propose the TreeLSTM network, which recursively encodes a tree by computing a node's representation based on its children using an LSTM. However, compared to the approach in this chapter the model is not parallelizable and thus slow. Yin and Neubig (2017) propose a model that generates code from natural language descriptions by following a rule-based approach over ASTs. Hu et al. (2018) linearize an AST and use a longer structure-based traversal as input for a regular sequence-to-sequence model to predict comments. LeClair et al. (2019) summarize source code by using two encoder networks, one that encodes the structure-based traversal of the AST and another that encodes the textual information in the sequence. Structure-based traversals have the major drawback, that they produce longer input sequences than the approach presented in this chapter (that only encodes the pre-order node sequence), which increases runtime. For the same purpose, Alon et al. (2019a)'s Code2seq model encodes paths between terminal tokens in an AST using a dedicated encoder-decoder architecture with attention. In contrast to Code2seq, our model encodes the full AST with all relative positions at once and additionally implicitly models the path between any two nodes. LeClair et al. (2020) and Fernandes et al. (2019) propose a structured summarization approach for code and natural language by adding a graph neural network on top of a sequence-to-sequence encoder. These approaches, however, utilize neither transformer networks nor structural losses or relative position representations.

Closest to the work in this chapter are is related work that study transformer models that encode syntax trees. Shiv and Quirk (2019) define absolute positional embeddings for regular trees and show that these can be used to leverage syntactic information. For this approach, the tree needs to be converted into a binary tree. This model is referred to as Absolute Tree Transformer. Concurrent to the work in this chapter, Kim

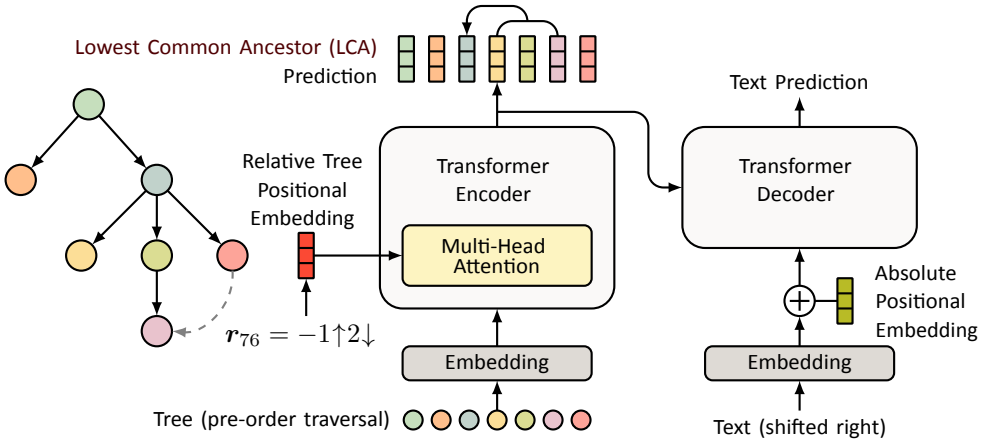


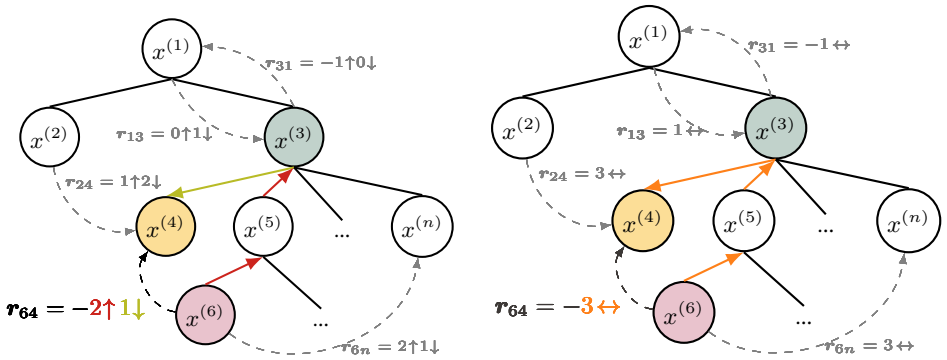
Figure 3.1: The Relative Structural Transformer (RST) architecture. The input to the transformer encoder (middle) is the pre-order traversal of the tree (left). Structural information about the tree is encoded with relative positional embeddings (red) that encode movements between nodes. The transformer decoder (right) is not altered and operates on sequences using a standard architecture. This figure is adapted from **Villmow** et al. (2021b) ©2021 IEEE.

et al. (2021) propose a similar approach to the one presented in this chapter, but utilize precomputed relative node positions and only apply them for scalar boosting of attention weights during self-attention. The Hierarchical Transformer (Nguyen et al. 2020) uses aggregation, masking, and hierarchical embeddings during self-attention to incorporate structure into transformers. Thereby, the authors change the attention mechanism, which negatively impacts performance (what will be show in Section 3.5.1 and Figure 3.4). In contrast, our method uses the widely adapted relative positional embeddings and otherwise follows a standard architecture. Also, none of these approaches use a structure-aware loss function to enforce the model to learn the structure of the tree.

3.3 Approach

As shown in Figure 3.1 this chapter aims to add a structural prior to transformers, specifically by encoding trees. Due to the focus on encoding trees and generating sequences, only the encoder is altered (middle), while the decoder remains unchanged. The approach starts with converting the tree into a pre-order traversed node sequence, including both terminal and nonterminal nodes. This turns the tree into a token sequence, making it compatible with the transformer encoder. Two key modifications are proposed:

1. In Section 3.3.1 relative positional embeddings (red in Figure 3.1) are defined that encode structural information with tree patterns (gray dashed arrow). This allows the transformer to recognize tree hierarchies.



(a) Movement Embeddings **(b)** Distance Embeddings

Figure 3.2: Hierarchical node relationships in a tree structure. In the context of nodes $x^{(4)}$ (yellow) and $x^{(6)}$ (purple), the LCA is node $x^{(3)}$ (blueish). Left side: To reach $x^{(4)}$ from $x^{(6)}$, two upward traversals are required (red, $2\uparrow$), followed by one downward traversal (green, $1\downarrow$). Thus the movement pattern is $\boldsymbol{r}_{64} = -2\uparrow 1\downarrow$. Right side: The distance between $x^{(6)}$ and $x^{(4)}$ is three steps (orange arrows). This is encoded in the distance pattern as $\boldsymbol{r}_{64} = -3\leftrightarrow$. In both methods going backwards in the pre-order traversal is indicated by a minus. Other patterns are shown in light gray.

2. The model is designed to encode structural information in its hidden states. To ensure this, in Section 3.3.3 a new loss function is introduced, forcing the model to identify the LCA of two nodes (top). This loss function is integrated with the regular translation loss as a weighted sum.

Also, please note that this approach could directly be used in encoder-only architectures, that aim to encode trees, and also in tree-generating decoders. However, both of these studies are left for future work.

3.3.1 *Relative Position Representations for Trees*

The approach developed in this thesis builds upon relative positional embeddings for sequences as introduced by Shaw et al. (2018), detailed in Section 2.3.2. The authors introduced the concept of relative position representations and distinguished between key $\boldsymbol{r}_{ij}^K$ and value $\boldsymbol{r}_{ij}^V$ embeddings. Their findings indicated that utilizing only key embeddings in the compatibility calculation in Equation (2.41) proved sufficient. Following this insight, this work implements the attention mechanism using Equation (2.41) from Shaw et al. (2018) and Equation (2.34) from Vaswani et al. (2017) (instead of using Equation (2.40) from Shaw et al. (2018)). Consequently, a single embedding $\boldsymbol{r}_{ij} \coloneqq \boldsymbol{r}_{ij}^K \in \mathbb{R}^{d_h}$ is used to encode the relative tree position between nodes i and j. Here, d_h represents the hidden dimension of the transformer attention head. The positional embedding is shared across all attention heads, but not across layers. The key difference to the work of Shaw et al. (2018) is that here, $\boldsymbol{r}_{ij}$ is based on the relative position of i and j in a tree, and not in a flat sequence.

The relationship between nodes in a tree is characterized by the number of upward and downward steps required to move from node i to node j, as illustrated in Figure 3.2. A matrix $\boldsymbol{M} \in \mathbb{R}^{n \times n}$, where n is the number of nodes in the tree, is used to record these steps. Starting from node i, one needs to ascend M_{ij} steps to reach the LCA $\text{lca}(i, j)$ of nodes i and j, and then descend M_{ji} steps to arrive at node j. Similar to Shaw et al. (2018), the distances in the movement matrix are clipped to a maximum path length k:

$$\boldsymbol{M} = \min(\boldsymbol{M}, k) \tag{3.1}$$

Two methods are examined for encoding positions of nodes in the tree into $\boldsymbol{r}_{ij}$. Both are derived from $\boldsymbol{M}$, but differ in the way the path is encoded.

Path length In the first method, the path from i to j via $\text{lca}(i, j)$ is represented by its length $l(i, j) = \min(M_{ij} + M_{ji}, k)$, ranging from $0, \ldots, k$. The positional embedding $\boldsymbol{r}_{ij}$ is derived from an embedding table $\boldsymbol{R} \in \mathbb{R}^{d_h \times (k+1) \times 2}$. The embedding is determined by $\boldsymbol{r}_{ij} := \boldsymbol{R}_{:,l(i,j),\mathbf{1}_{i<j}}$. For example, in the relation between parent node $x^{(1)}$ and child $x^{(3)}$ shown in Figure 3.2b, the path length pattern $1 \leftrightarrow$ indicates a parent-child step, while $-1 \leftrightarrow$ denotes a child-parent step. Note that the sign $(-)$ indicates the order of the nodes in the pre-order traversal.

Movement pattern The second method provides more granularity by separately encoding the upward (M_{ij}) and downward (M_{ji}) steps between the nodes. Here, the embedding table $\boldsymbol{R} \in \mathbb{R}^{d_h \times (k+1) \times (k+1) \times 2}$, contains twice as many embeddings as in the path length method, to account for both upward and downward steps. The encoding is defined by $\boldsymbol{r}_{ij} := \boldsymbol{R}_{:,M_{ij},M_{ji},\mathbf{1}_{i<j}}$. For instance, in the relationship between nodes $x^{(6)}$ and $x^{(4)}$, shown in Figure 3.2a, the movement includes two upward steps $(2\uparrow)$ to the LCA $x^{(3)}$, followed by one downward step $(1\downarrow)$ to the left $(-)$ to reach $x^{(4)}$, represented as $-2\uparrow 1\downarrow$.

While both methods model parent-child relationships explicitly, the movement pattern method offers greater expressive power than the path length method. For example, in the path length method (Figure 3.2b), $\boldsymbol{r}_{24} = \boldsymbol{r}_{6n} = 3 \leftrightarrow$ as both paths have the same number of steps and the same direction. In contrast, in the movement pattern method (Figure 3.2a), $\boldsymbol{r}_{24}$ is encoded as $1\uparrow 2\downarrow$, while $\boldsymbol{r}_{6n}$ is $2\uparrow 1\downarrow$.

3.3.2 *Efficient Computation*

Since the relative position representations are part of the forward pass, they need to be efficiently computable. The computation of tree position representations becomes straightforward given the matrix $\boldsymbol{M}$, since it is then only a matter of indexing into the embedding table $\boldsymbol{R}$. However, precomputing and storing $\boldsymbol{M}$ is not feasible for large datasets, as it has

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 2 & 3 & 2 & 2 & 2 \\ 1 & 1 & 2 & 2 & 3 & 3 & 2 \\ 1 & 1 & 2 & 2 & 3 & 4 & 2 \\ 1 & 1 & 2 & 2 & 2 & 2 & 3 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 1 & 0 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 & 0 & 0 & 1 \\ 3 & 3 & 2 & 2 & 1 & 0 & 2 \\ 2 & 2 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(a) Node incidence matrix $\boldsymbol{N}$, defined in Equation (3.2)

(b) Ancestral matrix $\boldsymbol{A}$, defined in Equation (3.4)

(c) Movements matrix $\boldsymbol{M}$, defined in Equation (3.5)

Figure 3.3: Matrices used to compute the movement matrix $\boldsymbol{M}$ for the tree from Figure 3.2.

a quadratic space complexity of $O(n^2)$. Instead, in the following it is demonstrated that $\boldsymbol{M}$, and thus $\boldsymbol{r}_{ij}$, can be efficiently computed through matrix operations from a linear sequence of the number of descendants of each node. For the tree in Figure 3.2, the stored descendant count sequence is $(6, 0, 4, 0, 1, 0, 0)$. This count can be precomputed in linear time and space.

The goal is to represent the tree with n nodes by a binary *node incidence matrix* $\boldsymbol{N} = \{0, 1\}^{n \times n}$, which denotes for each node its path to the root in the tree[3]. In the node incidence matrix, the element N_{ij} is set to 1 if node i is an ancestor of node j, and 0 otherwise. Note that $N_{ii} = 1$, because Section 2.1.2 defined each node as an ancestor of itself. An example of the node incidence matrix for the tree from Figure 3.2 is shown in Figure 3.3a. The matrix is formally defined as:

$$N_{ij} = \begin{cases} 1 & \text{if } j \in \text{ancestors}(i) \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

Since, the tree is represented as a pre-order sequence of nodes, $\boldsymbol{N}$ can be easily derived from the stored descendant count sequence (the size of each node's subtree) by populating $|\,\text{descendants}(j)| + 1$ rows in the j-th column of $\boldsymbol{N}$ with 1 starting from the diagonal:

$$N_{i,j} = \begin{cases} 1, & \text{if } i \in \{j, j+1, \ldots, j + |\,\text{descendants}(j)|\} \\ 0, & \text{otherwise.} \end{cases}$$

This can be done with native tensor operations in libraries like PyTorch or NumPy. The node incidence matrix has interesting properties that can easily be computed with tensor operations in these libraries. For instance, a vector with the depths of all nodes $\boldsymbol{d} \in \mathbb{N}^n$

[3]The node incidence matrix is analogous to the incidence matrix used in graph theory, albeit representing node-node connections instead of node-edge relationships.

can be computed by summing the rows of $\boldsymbol{N}$:

$$\boldsymbol{d} = \boldsymbol{N} \cdot \mathbb{1}_n \tag{3.3}$$

Additionally, the symmetrical *ancestral matrix* $\boldsymbol{A}$ can be derived from $\boldsymbol{N}$ (Andriantiana et al. 2018), which contains the depth of the LCA of a node pair $A_{ij} = \text{depth}(\text{lca}(i, j)) = \boldsymbol{d}_{\text{lca}(i,j)}$. Since the depth of the LCA is equal to the number of shared ancestors, the ancestral matrix can be computed by multiplying the node incidence matrix with its transpose:

$$\boldsymbol{A} = \boldsymbol{N}\boldsymbol{N}^\top \tag{3.4}$$

With $\boldsymbol{A}$, the movement matrix $\boldsymbol{M} \in \mathbb{N}^{n \times n}$ (Figure 3.3c) can be computed as follows:

$$\boldsymbol{M} = \boldsymbol{N} \cdot \mathbb{1}_{nn} - \boldsymbol{A} \tag{3.5}$$

After precomputing an array with the number of descendants of each node in linear time and space, $\boldsymbol{M}$ can be computed with parallel matrix multiplications efficiently on a standard GPU—which are optimized for these operations. These operations are executed with a cubic time complexity, $O(n^3)$, which is practical for input sizes common in NLP, such as sequences up to 1024 in length. This will be demonstrated in the experiments.

3.3.3 *Structural Loss*

Relative positional representations for trees, as introduced in the last section, provide the model with the ability to capture the structural relationships between nodes in a tree. However, the model is not explicitly trained to utilize the structural information encoded in the relative positional representations. This gap is addressed by introducing a structural loss, which encourages the model to retain the structural information within its hidden states in a useful manner. It operates on the principle of node similarity. Nodes in the same syntactical unit, like expressions in the block of an `IfStatement`, are considered similar, while those in different units are deemed dissimilar. The key element connecting the two nodes is their LCA, $\text{lca}(i, j)$, e.g., the `IfStatement` in the example. For the model to reflect this node similarity in its embeddings, it is trained to predict a node pair's LCA. Successfully predicting the $\text{lca}(i, j)$ indicates the model's understanding of syntactic dependencies, and thus the tree structure.

LCA prediction starts with the concatenation of $\boldsymbol{z}^{(i)}$ and $\boldsymbol{z}^{(j)}$, the transformers output embeddings of nodes i and j. This concatenated vector is processed through a linear layer with ReLU activation, producing an output vector $\boldsymbol{v}_{ij}$. The aim is to align $\boldsymbol{v}_{ij}$ closely with the LCA's output vector $\boldsymbol{z}^{(\text{lca}(i,j))}$. Next, $\boldsymbol{v}_{ij}$ is compared with the output embeddings from the transformer encoder, using the dot product as a similarity measure. A softmax layer then transforms these similarity scores into a probability distribution over all tree

nodes. The process is formally described as follows:

$$\boldsymbol{v}_{ij} = \text{ReLU}(\left[\boldsymbol{z}^{(i)} \oplus \boldsymbol{z}^{(j)}\right] \cdot \boldsymbol{W} + \boldsymbol{b}) \tag{3.6}$$

$$p_{\text{lca}}(a|i,j) = \text{softmax}\left(\boldsymbol{v}_{ij} \cdot \boldsymbol{Z}\right)_a \tag{3.7}$$

In these equations, $\boldsymbol{W} \in \mathbb{R}^{2d \times d}$ and $\boldsymbol{b} \in \mathbb{R}^d$ are the weight matrix and bias vector of the linear layer. The dimensionality of the transformer encoder's hidden layer is denoted by d, and $\boldsymbol{Z} \in \mathbb{R}^{d \times n}$ holds the stacked output embeddings $\boldsymbol{z}$ of the transformer encoder.

The softmax vector contains the probability $p_{\text{lca}}(a|i,j)$, representing the likelihood of each node a being the LCA of nodes i and j. The loss is computed as the negative log-likelihood across these node pairs.

$$\mathcal{L}_{\text{lca}}(\boldsymbol{\theta}) = -\sum_{i=1}^{n}\sum_{j=1}^{n} \log p_{\text{lca}}(\text{lca}(i,j)|i,j,\boldsymbol{z}) \tag{3.8}$$

Integration into Training

Sequence-to-sequence training occurs on input-output pairs $(\boldsymbol{x}, \boldsymbol{y})$. Here, $\boldsymbol{x}$ denotes the input sequence of a pre-order linearized tree, such as an AST or constituency parse tree. Correspondingly, $\boldsymbol{y}$ represents the desired output sequence, which may be a method name or documentation. In the context of training transformer sequence-to-sequence models end-to-end, a common approach is to use the label-smoothed cross-entropy translation loss, as detailed in Equation (2.17). This loss is combined with the structural loss, so that both losses are optimized simultaneously, where λ_{lca} controls the influence of the structural loss on the overall loss[4]:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_{\text{seq-smooth}}(\boldsymbol{\theta}) + \lambda_{\text{lca}} \cdot \mathcal{L}_{\text{lca}}(\boldsymbol{\theta}) \tag{3.9}$$

Sampling for Lowest Common Ancestor Prediction

The structural loss calculation for the LCA prediction in Equation (3.8) is based on all node pairs in a tree. As the number of node pairs exhibits a quadratic relationship with the tree's node count, this method becomes computationally demanding. To mitigate this, the loss is calculated on a subset of ϕ_{lca} node pairs (i,j) and their LCAs, sampled from the tree. Computing the loss only on a subset of node pairs is inspired by the negative sampling approach used in word embeddings (Mikolov et al. 2013b).

[4]Note that one could also combine the structural loss with other losses, e.g., for encoder-only models, but the focus of the experiments is on encoder-decoder models, hence the translation loss.

Algorithm 1: Efficient sampling of node pairs (i, j) and their LCA a for the structural loss calculation.

Result: Sampled node pairs (i, j) with their LCA a
Input: Tree $\mathcal{T}$ with node count n, number of samples ϕ_{lca} to draw
for $m \leftarrow 1$ **to** ϕ_{lca} **do**
  Sample node a from T with probability proportional to $|\text{descendants}(a)|$;
  if $|children(a)| \geq 2$ **then**
    Draw two distinct children c_1, c_2 of a;
    Sample i from c_1 and descendants(c_1) uniformly;
    Sample j from c_2 and descendants(c_2) descendants uniformly;
  else
    $i \leftarrow a$;
    Sample j from a's descendants;
  end
  Return node pair (i, j) and their LCA a;
end

A straightforward method would be to randomly select nodes i and j from the tree and calculate their LCA a= lca(i, j). However, this approach can become computationally demanding as the calculation of the LCA requires a traversal of the tree for each node pair. For large trees and many pairs, this becomes a bottleneck during training. To overcome these challenges this chapter proposes a sampling strategy shown in Algorithm 1 that first samples ancestor a and then nodes i and j from its descendants. The ancestor a is sampled from the nodes with a probability proportional to the number of its descendants $|\text{descendants}(a)|$. This method biases the sampling towards nodes with a larger descendant count, which are more likely to be common ancestors of two randomly chosen nodes, and leaf nodes, that have no descendants, are never sampled. This ensures that the selected node a will always have at least one child[5].

Nodes i and j are subsequently sampled from a's descendants. To ensure that i and j indeed have a as their LCA, the procedure varies depending on whether a has at least two children or only one:

1. If a has at least two children, two distinct children, c_1 and c_2, are first selected from a's children. Nodes i and j are then sampled from c_1 and c_2 and their descendants, respectively, guaranteeing that a is their LCA.

[5]Note that consequently, this algorithm will never sample $a = i = j$ where a is terminal. However, we found that this is not a problem in practice.

2. If a has only one child, i is assigned as a, and j is chosen from a's descendants. Selecting both i and j from a's descendants would result in a LCA that is a descendant of a, rather than a itself.

Note that due to the pre-order linearization of the node sequence, the descendants of a are positioned between a and $a + |\text{descendants}(a)|$. This sequence is already precomputed, since it is required for the computation of the movement matrix $\boldsymbol{M}$ (see Section 3.3.2).

3.4 Experimental Setup

This section describes the experimental setup used to assess the effectiveness of the proposed Rst model. This section first outlines the research questions that guide the experiments, the approach of this chapter to pre-process trees including the tokenization strategy, the tasks and datasets used for evaluation, and the hyperparameters and model configurations.

3.4.1 Research Questions

With the experiments the following research questions are addressed:

Research Question 3.1: *Does the approach described in this chapter improve the performance of transformer models compared to a sequential transformer baseline that is not aware of the tree structure?*

This question aims to assess whether the structural prior is beneficial compared to a standard transformer. To this end, the Rst model is compared to a strong standard sequential transformer baseline on three evaluation tasks and six standard datasets: method naming, code summarization, and machine translation. These are detailed in Section 3.4.3. Both models are built with the same architecture and hyperparameter setup, and trained on the respective task end-to-end. This chapter denotes the sequential transformer baseline as Transformer (no tree), which processes regular code tokens instead of trees.

Research Question 3.2: *How do the models trained in this chapter compare to the state-of-the-art on the method naming, code summarization, and machine translation tasks?*

To this end, both the Rst and the Transformer models are compared to the state-of-the-art on the respective tasks. The state-of-the-art models are selected based on the best reported results in the literature for each task. This includes a comparison of our sequential transformer baseline to other reported transformer baselines (Alon et al. 2019a), which allows comparing the impact of the proposed tokenization strategy (which is detailed in the next section). Additionally, this chapter compares on the method naming task against the absolute positional embeddings for trees as proposed by Shiv and Quirk (2019) for transformer models. This model is denoted as Absolute Tree Transformer, and for a fair comparison we will use the same model architecture and hyperparameter setup

as for the other models. Following Shiv and Quirk (2019) the maximum depth of the tree encoding is set to 64.

Research Question 3.3: *How much do the learned relative positional representations and the structural loss contribute to the performance?*

To this end this chapter conducts an ablation study to investigate the impact of both of our contributions—the relative positional representations and the structural loss—on the model's performance.

3.4.2 Pre-Processing Trees

As detailed in Section 2.1.1 it is common practice in NLP to use bottom-up tokenization algorithms such as BPE to minimize the vocabulary size and improve generalization. This had at the time of this work started to be used on code as well (Babii et al. 2019). However, BPE is not directly compatible with tree structures. This chapter proposes a novel approach and uses a combination of top-down and bottom-up tokenization strategies to preprocess trees. First, source code is converted into an AST using the tree-sitter library, see Section 2.1.2. After parsing the tree, this chapter uses the following code-specific BPE tokenization strategy on the tree's terminal nodes:

- First, tokens are split on camel case or snake case into the individual parts. This is a common practice in machine learning for code (Alon et al. 2019a). For example, the identifier `get50th_Percentile` is segmented into (`get`, `50`, `th`, `Percentile`).
- Then the BPE algorithm from Sennrich et al. (2016) is applied to each token individually, resulting in further segmented tokens like `get`, `50`, `th`, `Percent@@`, `ile`.
- During the BPE procedure numbers are segmented into individual digits, which transforms the token sequence into `get`, `5@@`, `0`, `th`, `Percent@@`, `ile`. This modification enables the representation of all numbers with just 20 tokens, and, hypothetically, allows the model to generalize better to unseen numbers.
- Finally, all string literals are standardized to a single token (`<STRING>`).

This tokenization process modifies the structure of the AST by breaking identifiers and other terminal nodes into several sub-tokens. When a terminal node $x^{(i)}$ is divided into sub-tokens $(t_1, \ldots, t_m)$, it is replaced by a new subtree. This subtree arranges the sub-tokens in a sequential chain, so that each sub-token t_{j+1} is a child of t_j for $j \in \{1, \ldots, m-1\}$. The parent of $x^{(i)}$ becomes the parent of t_1 and $x^{(i)}$ is removed. For instance, the identifier of the example is represented as a linear chain of nodes `get` ← `50` ← `th` ← `Percent@@` ← `ile`.

3.4.3 Tasks and Datasets

The model is evaluated end-to-end on three distinct tasks—method naming, code summarization, and neural machine translation—without using any pretrained models.

Method Naming

The prediction of a method's name from its body and signature is addressed in this task. It is commonly applied to three distinct datasets: Java-small, Java-med, and Java-large, which were introduced by Alon et al. (2019a). These datasets are comprised of Java files sourced from open-source projects on GitHub, divided into training, validation, and test sets at the project level.

The Java-small dataset includes approximately 700,000 methods from 11 projects, allocating 9 projects for training, one for validation, and one for testing. Java-med contains around 4 million methods from 1,000 projects, while Java-large encompasses about 16 million methods from 9,550 projects. For validation and testing, Java-med designates 100 projects each, and Java-large allocates 250 projects for validation and 300 for testing. The methodology for predicting method names follows the approach of Alon et al. (2019a) and Allamanis et al. (2016) and treats the prediction as a sequence of sub-tokens, split based on camel case and underscores. As Alon et al. (2019a), we report case-insensitive micro-averaged precision, recall, and F1-score over the target sequence.

Code Summarization

The code summarization task is focused on generating natural language descriptions for code snippets, typically derived from the first line of the code's documentation. The Javadoc format, for instance, defines the first sentence of the docstring as a concise and comprehensive description of a method's functionality (Oracle 2024). Evaluation of this task is performed using two primary datasets: FunCom and CodeSearchNet.

Introduced by LeClair and McMillan (2019), the FunCom dataset was curated from a vast corpus of 51 million Java methods (Lopes et al. 2010), undergoing a cleaning process to ensure the comments were in English and to filter out samples based on token count constraints in both the description and the method body. Specifically, samples were excluded if the description contained fewer than 3 or more than 13 tokens, or the method body exceeded 100 tokens. The refined FunCom dataset comprises 2.1 million Java method/documentation pairs, divided into training, validation, and test splits at the project level. Given that descriptions in the FunCom dataset are considerably longer than method names, averaging 7.6 tokens compared to 3, the BLEU score is employed to assess the accuracy of the predicted comment against the reference comment. For the FunCom dataset, both the overall BLEU score and individual n-gram precisions are

reported, computed using the same script as LeClair and McMillan (2019) for a consistent evaluation methodology.

The CodeSearchNet dataset, introduced by Husain et al. (2019), includes method-documentation pairs across six programming languages, focusing on code search as its primary task. This chapter uses a version of the CodeSearchNet dataset, specifically curated for code summarization by Feng et al. (2020). The evaluation makes use of the script provided by the authors of the filtered dataset, and reports smoothed BLEU score. To maintain consistency with CodeBert, splitting on camel or snake case is omitted which preserves the tokenization of Feng et al. (2020) after the removal of BPE. This chapter trains a joint model for all six programming languages, to promote knowledge transfer and improving model performance across languages.

Neural Machine Translation

To assess the model's capabilities beyond source code-related tasks, this chapter evaluates the model on neural machine translation in translating sentences between English and German. Subsequently, contrary to the source code-specific models evaluated in previous tasks, the model is compared with natural language processing models that have been used for this task. Specifically, comparisons are made with sequence-based baseline models such as Conv-Seq2Seq (Gehring et al. 2017), Transformer (Vaswani et al. 2017), and Dynamic Convolution (Wu et al. 2019), which process inputs as sequences of tokens to produce output sequences. Additionally, tree-based models that have been shown to outperform sequence-based models, including Tree2Seq (Shi et al. 2018) and the more recent Hierarchical Transformer (Nguyen et al. 2020), which encode constituency parse trees to generate sequences of tokens, are also compared using the tokenized BLEU-score. Further details on these approaches can be found in Section 3.2.

The Iwslt'14 dataset, consisting of 160k English-German sentence pairs, is utilized for training the models in an end-to-end manner without pretrained models (Cettolo et al. 2014). Following the methodology of Nguyen et al. (2020), 5% of the dataset is reserved for validation, and the combined datasets (`IWSLT14.TED.dev2010`, `dev2012`, `tst2010-tst2012`) are used for testing; and the Stanford CoreNLP parser (Stanford NLP Group 2018; Manning et al. 2014) is used to generate tree representations for the input data; model performance is assessed by averaging the 5 checkpoints that yielded the highest validation BLEU scores from the run with the highest overall validation BLEU. The Byte Pair Encoding (BPE) technique was applied to the terminal nodes of constituency parse trees using the same approach as outlined for source code in Section 3.4.2, albeit without splitting tokens on camel case and underscores.

3.4.4 Hyperparameters and Setup

The implementation builds on top of PyTorch (Paszke et al. 2019) and the fairseq library (Ott et al. 2019). The transformer architecture consists of 6 encoder and 6 decoder layers, 4 attention heads, d=512 dimensional hidden states/token embeddings, and 1024 dimensional feed-forward layers, with shared input and output matrices in the decoder. This setting has been widely used in the literature and is closely aligned with the work of Nguyen et al. (2020) (Hierarchical Transformer). It allows adopting most hyperparameters from Nguyen et al. (2020) and optimizing only those specific to infrastructure, the structural loss and relative positional representations.

The Adam optimizer (Kingma and Ba 2015) is configured with β_1=0.9, β_2=0.98, and a weight decay of 0.0001. An inverse square root learning rate schedule (Vaswani et al. 2017) is employed, with a warm-up phase of 4000 steps leading to a peak learning rate of $5e{-}4$. A dropout rate of 0.3 is applied to the model. The label-smoothed cross-entropy translation loss from Equation (2.17) is used with a smoothing factor of ϵ=0.1. Beam search is used for sequence generation, with a beam size of 5. Thereby, repeating n-grams longer than two tokens are prohibited.

The BPE models are trained individually for each task on the respective training set and consist of $16k$ sub-tokens for all code-related tasks. For the neural machine translation task, the vocabulary is limited to $10k$ sub-tokens. For the transformer token baseline, the same BPE preprocessing steps are applied to the tokens. The LCA loss, as described in Equation (3.6) in Section 3.3.3, samples $\phi_{\text{lca}} = \min(n, 50)$ pairs per sample. A grid search is conducted on the validation set to optimize hyperparameters, such as the batch size, the weight of the structural loss $\lambda_{\text{lca}} = \{0.05, 0.3, 0.6, 1\}$ in Equation (3.9), the choice between path length or movement pattern for relative positional representations along with the clamping value k, and the decision to train the model with a joined vocabulary for the encoder and decoder. For the path length and movement pattern, values $k = \{2, 3, 8, 16\}$ and $k = \{4, 6, 8, 16, 32\}$, respectively, are explored. Three runs with the best hyperparameters are performed and the model with the highest validation score is reported. A comprehensive list of all hyperparameters is provided in Table A.1.

3.5 Results

This section presents the results of the experiments conducted to evaluate the Rst model. It first compares the Rst model against the sequential transformer baseline and the state-of-the-art models, and then conducts an ablation study.

Model	Java-small			Java-med			Java-large		
	P	R	F1	P	R	F1	P	R	F1
Code2vec†	18.5	18.7	18.6	38.1	28.3	32.5	48.2	38.4	42.7
TreeLSTM†	40.0	31.8	35.5	53.1	41.7	46.7	60.3	48.3	53.6
ConvAttention†	50.3	24.6	33.1	60.8	26.8	37.2	60.7	27.6	38.0
Transformer (no tree)†	38.1	26.7	31.4	50.1	35.0	41.2	59.1	40.6	48.1
Code2seq†	50.6	37.4	43.0	61.2	47.1	53.2	64.0	55.0	59.2
Transformer (no tree)	48.5	45.9	45.9	57.5	57.1	56.2	66.2	63.8	63.9
Absolute Tree Transformer	47.2	45.6	45.0	59.3	57.9	57.3	66.6	64.2	64.3
Relative Structural Transformer (Rst)	**52.7**	**47.6**	**48.6**	**61.3**	**60.0**	**59.4**	**66.6**	**64.6**	**64.5**

Table 3.1: F1-score for method naming. Results marked with † have been reported by Alon et al. (2019a). Adapted from **Villmow** et al. (2021b) ©2021 IEEE.

3.5.1 Comparison against State-of-the-Art

> *Does the approach described in this chapter improve the performance of transformer models compared to a sequential transformer baseline that is not aware of the tree structure?* – RQ 3.1
>
> *How do the models trained in this chapter compare to the state-of-the-art on the method naming, code summarization, and machine translation tasks?* – RQ 3.2

This section aims to answer the research questions RQ 3.1 and RQ 3.2 by comparing the Rst and the baseline models against the state-of-the-art models on the method naming, code summarization, and machine translation tasks, which are discussed one by one.

Method Naming

In the task of method naming, the model demonstrates superior performance across all three datasets, as shown in Table 3.1. It consistently outperforms the state-of-the-art models, ConvAttention and Code2seq, by more than 6 p.p. in F1-score. Notably, even the sequential transformer baseline exceeds these models and other reported sequential transformers in F1-score and recall across all datasets, with variable precision outcomes that improve on larger datasets. This significant improvement can be attributed to the specific preprocessing and BPE tokenization approach (refer to Section 3.4.2). The ConvAttention and Code2seq models lack BPE, resulting in lower recall due to their inability to predict unknown tokens. This showcases the effectiveness of the proposed preprocessing pipeline and the use of BPE for tokenization in source code.

Furthermore, our method also outperforms a transformer that uses absolute tree embeddings (Shiv and Quirk 2019) to add a structural prior to the transformer, indicating the advantages of relative positional representations over absolute ones. This is discussed in

Model	B_1	B_2	B_3	B_4	BLEU
AttendGRU †	-	-	-	-	17.4
AST-AttendGRU ‡	37.1	21.1	14.3	10.9	18.7
Graph2Seq‡	37.6	21.3	14.1	10.6	18.6
Code2Seq‡	37.5	21.4	14.4	11.0	18.8
BiLSTM+GNN-LSTM‡	37.7	21.5	14.6	11.1	19.1
Code+GNN+BiLSTM-2hops‡	39.1	22.5	15.3	11.7	19.9
Transformer (no tree)	40.3	23.6	16.4	12.6	21.1
Relative Structural Transformer (RST)	**42.3**	**24.4**	**16.8**	**12.9**	**21.7**

Table 3.2: Code summarization on the FunCom dataset (java). We report cumulative BLEU-4 score, together with single n-gram scores up to 4 n-grams (B1, . . . , B4), evaluated with the script released along with the dataset. Results marked with † have been reported by LeClair and McMillan (2019), results marked with ‡ by LeClair et al. (2020). Previously published in **Villmow** et al. (2021b) ©2021 IEEE.

Model	Ruby	JS	Go	Python	Java	PHP	All
Transformer† (no tree)	11.2	12.0	16.4	15.8	16.3	22.1	15.6
RoBERTa†	11.2	11.9	17.7	18.1	16.5	24.0	16.6
CodeBert†	12.2	14.9	18.1	**19.1**	17.7	**25.2**	17.8
Transformer (no tree)	13.9	14.6	18.1	18.2	18.2	23.1	17.7
Relative Structural Transformer (RST)	**14.8**	**15.0**	**18.6**	17.9	**18.6**	23.8	**18.1**

Table 3.3: Code summarization results on the CodeSearchNet dataset. As Feng et al. (2020) we report smoothed cumulative BLEU-4 scores. Results marked with † have been reported by Feng et al. (2020). Previously published in **Villmow** et al. (2021b) ©2021 IEEE.

more detail in Section 3.5.2. It is worth noting, that the gap between our model and the sequential transformer baseline narrows on larger datasets, suggesting that structural information becomes less important as more data becomes available. This observation highlights the influence of structural information in scenarios with limited data. It can significantly improve model performance.

Code Summarization

The experiments on code summarization further confirm the advantages of incorporating structural information. The model performs better than state-of-the-art models on the FunCom dataset by 1.8 p.p., and exceeds the sequential transformer baseline by 0.6 p.p. in BLEU score, as shown in Table 3.2. This confirms that the contributions allow the transformer to utilize structural information to its benefit. Similar improvements were observed in Table 3.3 on the CodeSearchNet dataset. The RST model outperformed the sequential transformer baseline by 0.44 p.p. in BLEU score, while the sequential transformer baseline also exceeded the performance of reported transformer baselines. It is worth noting that the reported transformer baselines on this dataset use a non-source code specific BPE from the RoBERTa model (Feng et al. 2020; Liu et al. 2019). The

results indicate that the BPE from the RoBERTa model may not be the best choice for source code.

When compared to the pretrained state-of-the-art CodeBert and RoBERTa models on this task, it is evident that the Rst, which is trained end-to-end without pretraining, outperforms the state-of-the-art on most programming languages, except for Python and PHP. This finding highlights the effectiveness of structural information and the approach.

Machine Translation

In the domain of machine translation, the model performs better than the sequential transformer by up to 1.1 p.p. in BLEU score, as shown in Table 3.4. Although the model outperforms other recent strong models, it falls behind the Hierarchical Transformer. Nevertheless, it is worth noting that the model achieves competitive results with significantly lower runtime and memory consumption than the Hierarchical Transformer, as shown in Figure 3.4. Efficiency is especially important for code datasets, where the Hierarchical Transformer model struggles with memory limitations due to the typically longer length of source code compared to natural language sentences, preventing the application of the model with reasonable batch sizes on the code tasks. The Rst model, on the other hand, performs similarly to the original sequential transformer (Transformer), while adding only a negligible overhead. Overall, the results indicate, that the model is applicable to various types of trees ranging from natural language to source code. It is able to utilize the structural prior from the data to improve over a sequential transformer baseline.

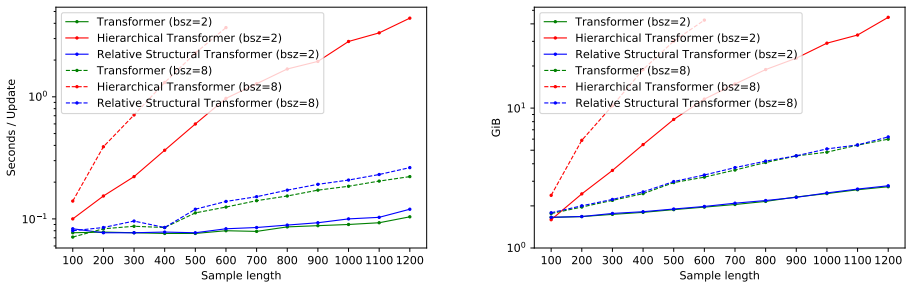
Model	En-De	De-En
Tree2Seq†	24.0	30.0
Conv-Seq2Seq †	24.8	30.3
Transformer† (no tree)	28.4	34.4
Dynamic Convolution†	28.4	34.7
Hierarchical Transformer†	**29.5**	**36.0**
Transformer (no tree)	28.3	34.6
Rst	29.4	35.3

Table 3.4: *Machine translation* on the IWSLT'14 dataset. We report the tokenized cumulative BLEU-4 score. Results marked with † have been reported by Nguyen et al. (2020).

3.5.2 *Ablation Study*

> *How much do the learned relative positional representations and the structural loss contribute to the performance?* – RQ 3.3

This section presents a discussion of the performance of various model variants on the Java-med dataset, as illustrated in Table 3.5. Models are compared based on including or excluding structural information by using relative positional representations and applying structural loss. The results validate the design choices of the final model by showing the importance of both contributions to model performance on the method naming task. In



(a) Speed comparison.

(b) Memory usage.

Figure 3.4: The figure illustrates comparisons of memory usage (measured in GiB) and speed (seconds per iteration) relative to sequence length, while keeping the batch-size (bzs — samples per batch) constant. It compares a sequential transformer, applicable as well to the Absolute Tree Transformer (Shiv and Quirk 2019) (due to pre-computable absolute positional representations), with the Hierarchical Transformer (Nguyen et al. 2020) using its reference implementation, and the RST that integrates all enhancements. These comparisons were made under uniform conditions: same model architecture (layers, dimensions, and vocabulary), datasets with uniform sample lengths, all executed on a single Quadro RTX 8000 (48GiB) GPU.

this ablation study, the model is trained with sinusoidal absolute positional representations for the input sequence when relative positional representations are excluded.

No Structural Information Linearizing the AST without incorporating any structural information significantly reduces the effectiveness of the model. There was a 4.3 p.p. decrease in F1-score compared to the best-performing model. This approach requires the model to infer the structure of the AST without explicit information. This is not always possible, because of ambiguous situations where multiple valid trees exist. Additionally, since the model is not required to learn the tree structure, it may take suboptimal shortcuts to predict the method name, potentially resulting in overfitting. Overall, using a sequential transformer model is more effective than linearizing an AST without any structural information.

Only Structural Loss Incorporating structural loss as an auxiliary task, without directly providing structural information, results in a slight reduction of 0.7 p.p. in F1-score compared to the best model. When no structural information is provided as input, but the model is trained with the structural loss, the model is trained to infer potential tree structures through its understanding of the generating grammar and to retain the structural details in the output representations z to predict the LCA. However, similar to when no structural information is available, this method may encounter difficulties in situations where there are multiple valid tree structures, which could lead to inaccurate predictions of the true tree. The fact that the sequential transformer baseline was outperformed suggests that the structural loss is a promising approach to guide the model in learning the

Relationship Type / Model	k	γ_{lca}	Precision	Recall	F1-score
Transformer	-	-	57.5	57.1	56.2
Absolute Tree Transformer	-	-	59.3	57.9	57.3
-	-	-	56.2	56.2	55.1
-	-	0.3	60.8	59.3	58.7
Movements	2	-	60.5	58.8	58.4
Movements	2	0.05	61.1	59.2	58.9
Movements	2	0.3	**61.3**	**60.0**	**59.4**
Path-Length	8	0.3	60.6	59.4	58.8

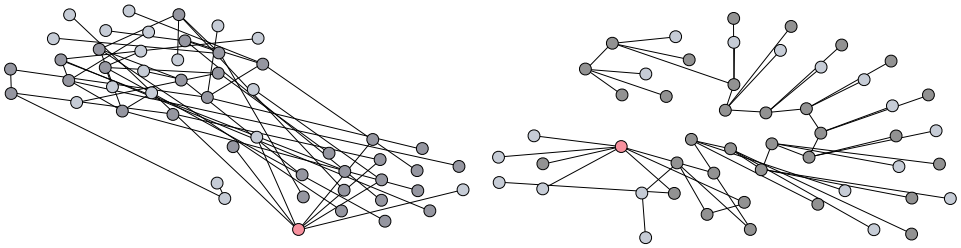
Table 3.5: Ablation study on Java-med. All models are trained on linearized ASTs. Adding a structural loss or relative positions improves performance. Adding both gives best results. Previously published in **Villmow** et al. (2021b) ©2021 IEEE.

tree structure. However, it may not be sufficient on its own due to the aforementioned ambiguities.

Only Relative Positional Representations When a structural prior is provided to the model in the form of relative positional representations, the model performs slightly worse than when using structural loss alone, with a 1.0 p.p. decrease in F1-score compared to the best model. In this variant, the model can use the structural prior to predict the method name, but it is not required to learn and retain the tree structure since it is provided in the form of relative positional representations. Furthermore, there is no need to reason about the tree structure, which may help prevent underfitting. This approach outperforms both the sequential transformer baseline and absolute tree embeddings (Absolute Tree Transformer). This indicates that the relative tree patterns are more expressive than the absolute tree embeddings.

Both Contributions The combination of both structural loss and relative positional representations results in the highest performance, achieving an F1-score of 59.4%. Also, increasing the share of structural loss further improves the performance. Although this model is not required to learn the tree structure from scratch, it must incorporate the structural prior into its output representations to predict the LCA effectively.

This is visualized in Figure 3.5, where Figure 3.5b shows that using the structural loss leads to transformer output representations that are more closely aligned with the tree structure, e.g., the representation of an `IfStatement` with child-expressions. The model trained without structural loss tends to group similar tokens, such as multiple `IfStatement` nodes, regardless of their hierarchical relationships in the tree (see Figure 3.5a). This highlights the effectiveness of incorporating structural loss to capture the essence of node similarity, as explained in Section 3.3.3.



(a) Without structural loss.

(b) With structural loss.

Figure 3.5: Visualization of z, the transformer encoder's output representations, projected to lower-dimensional space using t-SNE for the same input. Both models are trained with relative positional representations for trees, the model on the right additionally includes the structural loss. It can be seen, that the model trained with the structural loss in (b) reflects the tree structure much stronger in its output representations. In contrast, the model on the left in (a) groups similar tokens, such as nonterminal tokens (dark gray) in the lower right corner, regardless of their position in the tree. Figure adapted from **Villmow** et al. (2021b) ©2021 IEEE.

3.6 Conclusion and Future Work

This chapter introduced the Relative Structural Transformer (Rst), a novel approach to integrate structural information from trees into transformer models. It provided the transformer with an inductive bias for tree structures, by extending relative positional embeddings to encode the hierarchical relationships between nodes in trees. This was complemented by a novel structural loss function that predicts the LCA of node pairs, which encourages the model to retain and also utilize the structural information within its hidden states. Also, the chapter demonstrated efficient computation methods for these relative positional embeddings.

The approach was validated on three sequence-to-sequence tasks: method naming, code summarization, and machine translation. The experimental results showed that Rst model outperforms state-of-the-art models on the method naming task by 6 p.p. in F1-score and achieves competitive results on the other tasks, while always outperforming the sequential transformer baseline. An ablation study confirmed the individual contributions of the relative positional embeddings and the structural loss to the model's performance. Specifically, this chapter demonstrated that the combination of both contributions led to the best results. The findings of this chapter suggest that integrating structural information directly into the model architecture can significantly enhance performance on tasks that involve hierarchical structures. Also, this is not limited to source code, but can be applied to various types of trees, including natural language. Moreover, the efficiency of Rst in terms of memory and computational resources makes it a viable alternative compared to other models such as the Hierarchical Transformer.

A possible direction of future work could be to investigate the benefit of the proposed relative tree positional embeddings and the structural loss in transformer decoder models, for example in the context of autoregressive prediction of trees, such as in syntactic parsing or program synthesis. However, it is important to note that the impact of structural information decreases as the dataset size increases. This observation indicates that structural information can be useful in situations with limited data, but may become less necessary when models are pretrained on large datasets. The next chapter will study how to include structural information during pretraining.

I accept chaos, I'm not sure whether it accepts me.

— Bob Dylan, 1965

4

Structural Pretraining Tasks for Generative Transformer Models

Structural information can increase transformer model's code understanding capabilities. This has been demonstrated in the last chapter, which introduced the Rst model. Rst directly encodes code as an AST, and thus adds a structural prior to the model. To do so, the last chapter demonstrated that the transformer architecture is able to encode trees, when provided with appropriate positional encodings and a novel structural loss to enforce the model to represent code's syntactical structure in the latent space. This tree-based encoding led to improved performance over a regular token-based transformer model (on method naming and code summarization), an effect that was strongest in situations with limited data. The model was trained end-to-end on supervised tasks. However, the experiments showed that the additional benefits decreased when trained on larger datasets (more than 16M samples). This indicated that Rst's structural prior may

Work on this chapter has been done in 2021 as preliminary work to **Johannes Villmow** et al. (2022). Addressing Leakage in Self-Supervised Contextualized Code Retrieval. In *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*. International Committee on Computational Linguistics, pp. 1006–1013 and **Johannes Villmow***, Viola Campos*, Jean Petry, Amine Abbad Andaloussi, Adrian Ulges, and Barbara Weber (2023b). How Well Can Masked Language Models Spot Identifiers That Violate Naming Guidelines? In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*. IEEE, pp. 131–142.

Even though the model found application, the approach has been only briefly outlined in the above works. This chapter contains a more detailed description of the model and the pretraining tasks and provides plenty of additional experiments that have not been included in the above works.

be less important when large-scale data is available and also a token-level model will learn the structure from the data, given enough data.

Since the work on RST (which took place in 2018/2019), NLP research has moved on to even larger datasets, and pretraining approaches have become a stronger focus of research. This holds for code too: Situations in which large amounts of data are available are common in pretraining, where models are trained on extensive corpora of data before being fine-tuned for specific tasks. This chapter aims to investigate how the structural aspects of code can be used in the pretraining of transformer LMs, without the need for architectural modifications. It discusses how pretraining can be used to incorporate structural information into the transformer model. Compared to the previous chapter the ASTs are not directly used as an input to the model, but rather to construct challenging input/output examples for self-supervised pretraining. This makes it possible to train a regular transformer encoder-decoder model without any architectural modifications.

4.1 Introduction and Motivation

Since, 2018 pretraining has become a standard practice in NLP (Radford et al. 2018; Devlin et al. 2019; Lewis et al. 2020b). Contextualized transformer models are pretrained on extensive corpora of unlabeled data before being fine-tuned for specific tasks. This approach has proven effective across a broad spectrum of NLP tasks, and pretrained models are now the state-of-the-art on many NLP benchmarks (Wang et al. 2019). The alternative is to train a supervised model end-to-end on a task (as in Chapter 3), which requires a substantial amount of annotated data to achieve a sufficient performance, because of the complex nature of language. Typically, in NLP, generative *pretraining tasks* are used to teach a statistical *language model* general language understanding capabilities by learning the structure and grammar of the language, and capturing word interactions, as detailed in Section 2.4. This holds until 2024, at the time of writing this thesis, the size of the transformer models as steadily increased and the largest instruction-based models, such as ChatGPT, are initially trained with a self-supervised pretraining. After pretraining, much smaller annotated fine-tuning datasets can be used to achieve a better task-specific performance (Devlin et al. 2019). Thereby, pretrained transformer models remain architecturally mostly the same, often only a new task-specific output layer is trained. What makes the transformer architecture intriguing for pretraining is that the model can process all tokens in parallel (compared to LSTMs that process tokens sequentially). This allows to use much larger models and much larger pretraining datasets. Throughout this thesis all pretrained transformer language models will be referred to as Language Models (LMs)[1].

[1] The architectures utilized in this thesis have sometimes been called *Large Language Models* (LLMs) (Shanahan 2024), sometimes *Pretrained Language Models* (PLMs) (Zhao et al. 2023).

LMs are often trained by denoising a corrupted or partial input sequence. Section 2.4 describes the most common pretraining tasks used in NLP and how they are applied to transformer models. This section provides a brief summary. The specific pretraining task chosen for a model depends on its architecture. For encoder models like Bert and RoBERTa (Liu et al. 2019), tasks such as MLM combined with next sentence prediction are used. MLM masks and replaces random tokens in the input sequence and trains the encoder to recover the original tokens. Devlin et al. (2019) utilizes this task along with next sentence prediction as an auxiliary task, in which the input is constructed of two sequences, for which the model must predict whether these sequences follow each other in real-world training text. However, Liu et al. (2019) found this auxiliary task negligible or even deteriorating for downstream performance. Another option for encoder models is to use a discriminative training objective, as in Electra (Clark et al. 2020) or CodeBert (Feng et al. 2020), where a discriminator is trained to predict whether a token has been replaced by a generator model. For decoder models such as the GPT variants (Radford et al. 2018; Radford et al. 2019; Brown et al. 2020), causal/autoregressive language modeling is used, i.e., the model is trained to predict the next word, given all previous words. For generative encoder-decoder models like t5, short span prediction has been found to be an effective pretraining task (Raffel et al. 2020). This task replaces short spans between 3 and 5 tokens with sentinel tokens, which are concatenated in an output sequence, i.e., `MARKER1 SPAN1 MARKER2 SPAN2`. It is essentially a combination of a masking task in the encoder and an autoregressive prediction of a task-specific output sequence in the decoder. Raffel et al. (2020) demonstrated that span prediction leads to a good performance on a wide range of NLP tasks with relatively short output sequences.

All the above pretraining strategies have been applied on natural language. Pretrained models for source code have been developed similarly to those in NLP, with tasks such as MLM, span prediction, but also discriminative objectives (Feng et al. 2020; Wang et al. 2021b). Despite the effectiveness of these pretraining objectives in NLP, it is arguable whether they exploit the structural and semantic information present in code to its full potential. Moreover, Lachaux et al. (2021) were the first to claim that MLM is ineffective for code, as many tokens can be easily predicted without understanding the semantics of the code. For instance, many programming language keywords, syntax, and whitespace tokens only require that the model learned the simplest patterns of its underlying grammar (think of keyword `import`, which appears at the beginning of many Java/Python files' headers). No semantic understanding is necessary, which is less frequent in NLP. This effect is visualized in Figures 4.1a and 4.1b: one can see that most of the masked tokens are not particularly challenging to predict. Also, even when MLM eventually masks identifiers or function names, another occurrence of the identifier often remains unmasked in the surrounding context. Then the LM simply copies the most likely identifier from the context. This is a drastically easier task than predicting a viable name altogether, which

would require an understanding of the semantic role of an identifier. We hypothesize that this weakens the learning signal for the LM. This raises the overall research question of this chapter, whether LMs benefit from pretraining tasks that better address the unique characteristics of source code.

Therefore, this thesis investigates the use of Abstract Syntax Trees (ASTs) to construct input/output examples for denoising. This chapter refers to pretraining tasks that use the AST to construct input/output examples as *structural* pretraining tasks, while the tasks that mask random tokens or spans of a sequence are referred to as *regular* pretraining tasks. Some related work has proposed such tasks. For instance, the task *identifier deobfuscation* proposed by Lachaux et al. (2021), utilizes the AST to pick some random identifiers and mask *all* occurrences of an identifier with the same sentinel token. A sequence-to-sequence model aims to recover the original identifier names from the obfuscated code. This is a challenging task, not only for LMs but also for humans: just imagine yourself comprehending obfuscated source code. Research has confirmed that identifiers play an important role in comprehending code, as they enable reading the code top-down, during which identifiers provide semantic cues for the developer (Brooks 1983). This process can be severely hindered by obfuscated code, as shown by Fakhoury et al. (2020). Hence, inferring the original identifier names from obfuscated code is a challenging task that requires understanding the semantics of the program (Lachaux et al. 2021), because—opposed to MLM, causal language modeling, and span prediction—*all* occurrences of an identifier are hidden. This has the benefit, that since the identifier is only predicted once for all occurrences, the output sequences are short, which speeds up training. However, while the experiments of Lachaux et al. (2021) suggest that identifier deobfuscation trains an effective code encoder, its performance on code generation tasks leaves room for improvement[2].

Parallel to the work presented in this chapter, the CodeT5 model was proposed (Wang et al. 2021b). Wang et al. (2021b) adopt the T5 methodology from Raffel et al. (2020) and propose to train a multi-task encoder-decoder transformer on five tasks: identifier deobfuscation, code summarization (code to text), code generation (text to code), identifier tagging (binary classifier for identifiers), and short span masking. Thereby, the authors use not only unimodal, but also bimodal supervised data in form of comment-function pairs and frame all tasks as a sequence-to-sequence problem, allowing them to train one model for all tasks with the same loss. This is a combination of structural pretraining tasks (identifier deobfuscation and tagging) with regular pretraining tasks (short span masking) and also supervised bimodal data. While the combination of these tasks shows more variety than single-token masking, it can be argued that their suitability to instill semantic code understanding into a model is still limited: First, we argue that short span

[2] The authors try to address this problem by training the model together with a denoising auto-encoding task in which random tokens are removed and/or shuffled and the model is tasked to recover the original sequence. However, this task has the same limitations as regular MLM.

masking with an average span length between 3 and 5 tokens suffers from the same pitfalls as MLM, which—as described above—often creates "easy" targets. This is visualized in Figures 4.1c and 4.1d. Additionally, we find that identifier tagging is not challenging for an LM, it simply needs to learn the grammar rules for identifier positions, which does not require semantic code understanding. Furthermore, even though leveraging bimodal data for better alignment of natural language and code has been proven to be effective for many code-LMs (Feng et al. 2020; Wang et al. 2021b), it requires manual or semi-supervised data acquisition. Therefore, this chapter combines/extends the above approaches to investigate structural pretraining of code transformers further.

4.1.1 Contributions

This work continues the line of research on structural pretraining tasks for source code by introducing a novel structural pretraining task for encoder-decoder models called *tree-based span selection*, which selects spans for masking based on the code's AST. We argue that this task is much better suited for code than the short span masking used in CodeT5 and other models. The task samples one or more nodes from the AST and replaces the tokens in the input sequence that belong to the nodes with sentinel tokens. The decoder predicts the combined output sequence—as in the T5 model described above. Using the AST to select small sub-blocks of code leads to the removal of syntactic segments, such as comments, the conditions of if-statements, loops, or even full methods. Compared to regular random span masking, this approach is much more likely to produce challenging and contextually rich training examples. The task can be seen as a generalization of many code understanding tasks, such as code summarization and method naming (for task descriptions see Section 3.4.3). For example, in code summarization, the code-documentation pairs are typically bootstrapped from a method's docstring. This task is emulated with tree-based span selection when a comment node (containing a docstring) in the AST is selected. Note that this is self-supervised.

Additionally, since the input length of the transformer model is limited, code files need to be truncated, but traditional truncating methods are not particularly well suited for source code. This chapter will present a technique called tree-based file truncation that makes use of tree-based span selection to truncate code files to a maximum length by extracting large subtrees. This is similar to the folding feature available in code editors, where a developer can hide details within the code (e.g., the body of a method) to focus on the overall structure (e.g., to see which methods are available in a class). The technique outputs a context-rich code file in which parts are folded, along with the extracted parts as individual samples.

The main contribution of this chapter is a novel multi-task pretraining strategy for encoder-decoder transformer models. It combines the novel tree-based span selection task with

other (structural and regular) pretraining tasks from the above works, focusing on those that supposedly foster a strong semantic code understanding. The resulting model is called SyntaxPT. Thereby, the tasks are implemented in a dynamic pipeline, that allows to extend and improve identifier deobfuscation from Lachaux et al. (2021) by probabilistically sampling a corruption rate instead of using a fixed rate, obfuscating also method calls, uncovering some occurrences to facilitate more reasoning, and randomize the masking order. To this end, this chapter introduces and open-sources the tensortree library (**Villmow** 2021), that has been specifically implemented to work with pretokenized trees as working with PyTorch tensors (Paszke et al. 2019).

This chapter specifically wants to investigate whether the structural tasks improve the resulting models' code understanding capabilities even when large-scale training data is available. To this end, training is conducted on a large dataset of 30 million code files in 16 different programming languages from GitHub. SyntaxPT is to the best of the author's knowledge, the first LM trained with identifier deobfuscation on that many programming languages. The chapter compares this model to a baseline trained with regular pretraining tasks on several code understanding downstream tasks from CodeXGlue (Lu et al. 2021). The results demonstrate that structural pretraining significantly outperforms regular pretraining on most tasks.

In summary, the key contributions of this chapter are:

- Introduction of a novel self-supervised structural pretraining task called *tree-based span selection*, which selects appropriate spans for masking based on the program's AST to produce more challenging examples than short span masking.
- Extension and improvement of the structural pretraining task of *identifier deobfuscation* proposed by Lachaux et al. (2021), and training the first LM with this task on 16 different programming languages.
- Introduce a novel technique called tree-based file truncation that truncates code files to a maximum length by extracting large subtrees.
- Present the tensortree library that enables developers to work with trees in PyTorch. It is used to implement the structural pretraining tasks (**Villmow** 2021).
- Introduction of a novel multi-task pretraining approach for encoder-decoder LMs that combines structural with regular pretraining tasks and is trained only on unimodal data (i.e., code files) in a self-supervised fashion.
- Demonstration that structural pretraining leads to better code understanding capabilities compared to regular pretraining tasks, as evaluated on five CodeXGlue tasks.

- Outperforming the state of the art on four out of five CodeXGlue tasks by a significant margin.

4.2 Related Work

An overview over the most common pretraining tasks for transformer models in NLP has been given in Section 2.4. This section will only briefly focus on related work that has been covered before, and mostly discusses work that addresses pretraining for code-LMs, until the end of the experiments in beginning of 2022.

4.2.1 Pretraining Strategies in Natural Language Processing

In the field of NLP, pretraining transformer models has become a standard practice to enhance performance on various downstream tasks. Early approaches like ELMo (Peters et al. 2018) pretrained an LSTMs on causal language modeling and used the hidden states as features for downstream tasks. The most common language models have already been detailed in Section 2.4, such as the transformer decoder-based GPT model, and the encoder-based Bert model. Liu et al. (2019) demonstrate that with BPE, masking single independent tokens often keeps parts of the word uncovered, which has been found to render training less challenging (which is similar to identifiers in code as discussed above). Subsequent research focused on optimizing hyperparameters and training strategies. RoBERTa (Liu et al. 2019) improved upon Bert by adjusting model depth and learning rates and also employ full-word masking to always mask complete words. Albert (Lan et al. 2020) reduced the number of parameters through parameter sharing, and Electra (Clark et al. 2020) introduced a more sample-efficient pretraining task by replacing the MLM objective with replaced token detection using a discriminator.

Even though some approaches, such as Ciniselli et al. (2022) investigate utilizing Bert as a generative model, mostly encoder-decoder architectures and training strategies have been proposed for sequence-to-sequence tasks. Dong et al. (2019) proposed *UniLM*, a unified transformer model that supports unidirectional, bidirectional, and sequence-to-sequence pretraining through specific masking schemes. Bart (Lewis et al. 2020b) introduced a denoising autoencoder that reconstructs the original text from corrupted input using arbitrary noising functions. While Bart's setup provides freedom in the choice of the noising function, it requires generating the full input sequence, which is computationally demanding. Raffel et al. (2020) proposed t5, which uses a single encoder-decoder transformer for multiple sequence-to-sequence tasks, such as small span prediction. This reduces the size of the output sequence and speeds up training, compared to the approach of Bart. t5 has been shown to achieve state-of-the-art performance on a wide range of NLP tasks. Similarly, Pegasus (Zhang et al. 2020) proposed Gap Sentences

Generation (GSG) for abstractive summarization, where full sentences are masked, and the model is tasked with generating the missing sentences.

These pretraining strategies have significantly advanced the capabilities of NLP models, but the datasets of this generation of models are typically limited to natural language text and the tokenizer does not consider unique characteristics of source code, such as an extensive use of whitespace, which restricts their application to source code.

4.2.2 Language Models for Source Code

The application of language models to source code has gained considerable attention, aiming to improve tasks like code completion, summarization, and translation. Early efforts in code completion utilized statistical models such as n-grams (Hindle et al. 2012; Allamanis and Sutton 2013), and cached n-grams for improved localization (Franks et al. 2015; Tu et al. 2014; Hellendoorn and Devanbu 2017). Graph-based statistical language models were also explored (Nguyen and Nguyen 2015). With the rise of deep learning, neural network models were applied to code, including RNNs (White et al. 2015; Raychev 2016; Li et al. 2018) and LSTMs (Dam et al. 2016). These models captured longer-range dependencies than n-grams models, and improved code completion performance.

With the introduction of the transformer architecture and pretraining strategies in NLP, these models and strategies were adapted to source code. Kanade et al. (2020) introduced CuBert, and pretrained a Bert model on a large corpus of 7.4 million Python files from GitHub. The model was evaluated on code understanding tasks but focused on a single programming language. Feng et al. (2020) investigated adapting the RoBERTa model to code, by further pretraining it on functions from CodeSearchNet using the MLM objective. The authors show that the model outperforms the original RoBERTa model on the CodeXglue benchmark, which is why it is used as a strong baseline in this thesis. The model is referred to as RoBERTa-code. However, the authors did improve the model by adding a discriminative replaced token detection objective to MLM, as in the Electra model. Their proposed CodeBert model is jointly pretrained on bimodal code and natural language comment pairs, to learn the alignment between code and natural language.

Decoder-only models have also been explored for code. CodeGPT (Svyatkovskiy et al. 2020) used the GPT-2 architecture and trained on 1.2 billion lines of code in multiple programming languages. It targeted code completion and synthesis tasks and was integrated in a tool called IntelliCode Compose within Visual Studio. CugLM (Liu et al. 2020) built upon UniLM's architecture, with a variety of pretraining tasks, including MLM, next code segment prediction, and causal language modeling.

Encoder-decoder models have shown promise in capturing both the understanding and generative aspects required for code-related tasks. PLBART (Ahmad et al. 2021) adapted the BART approach to source code pretrained the model on a large-scale dataset comprising 470M Python and 210M Java functions, and 47M natural language posts from StackOverflow. The model is pretrained with a denoising autoencoder objective, that reconstructs the original code snippet from a corrupted version. The authors employ single token masking, token deletion, and span masking with span lengths sampled from Pois(3.5) as corruption strategies. CoTexT (Phan et al. 2021) fine-tuned the T5 model on both unimodal (code-only) and bimodal (code and text) data and investigated a multi-task fine-tuning on the CodeXGlue benchmarks.

Contrary to the approach in this chapter, most of these models use bimodal data to learn an alignment between code and natural language. This requires supervised data or reling on large-scale code-comment pairs. Moreover, they may not fully exploit the structural properties inherent in source code. All the aforementioned strategies often target easy targets, which may result in less effective pretraining on code, as outlined in the introduction. For code understanding tasks that do not require generation, such as defect detection or code retrieval, encoder architectures are especially useful due to their ability to bidirectionally compare input tokens during attention. Decoder models, on the other hand, such as GPT-2, are limited to attending to previously processed tokens, while their ability to produce sequences of unknown length is necessary for sequence generation tasks, such as code summarization. The encoder-decoder architecture combines the advantages of both and has been shown to be versatile across a wide range of tasks (Raffel et al. 2020). Thats why this chapter focuses on a encoder-decoder model.

4.2.3 Structural Pretraining for Source Code

The idea that the structural properties of source code can be leveraged for pretraining has been an active area of research at the time of the experiments for this chapter. Jiang et al. (2021) introduced TreeBERT, which represented the AST as a set of composition paths—similar to Code2seq—and introduced a tree-based MLM and node order prediction tasks on the structural input, but also generated the original code sequence in the decoder. However, these paths lead to longer input sequences and increased training times. GraphCodeBert (Guo et al. 2021) improved upon CodeBert by incorporating two structural pretraining tasks that utilize the AST to (1) predict data flow connections between variables, i.e., from which variable is another variable derived, and (2) align the representations of data flow nodes with variables in the code. GraphCodeBert is pretrained on bimodal data. SynCobert (Wang et al. 2021a) incorporated multi-modal data, including code, comments, and AST representations. Pretraining tasks included binary identifier tagging (is there an identifier at this position?), edge prediction in the AST, and multi-modal contrastive learning to obtain code representations that feature

both symbolic and syntactic properties. UniXcoder (Guo et al. 2022) employed an encoder architecture adaptable for code generation tasks, with masking strategies that considered the AST and code tokens. It focused on cross-modal pretraining with code comments and AST information. The most similar work to the one in this chapter is CodeT5, which has been detailed in the Introduction.

However, contrary to the approach in this chapter, many of these models have the drawback that they require ASTs as input to the model at inference time, which is not always available (e.g., when the code is incomplete or syntactically incorrect). Moreover, the approach in this chapter remains architecturally standard, and uses a standard encoder-decoder transformer without any modifications. Also, unlike many of the aforementioned approaches that utilize bimodal data, the approach of this chapter pretrains solely on unimodal code data. This makes the training and input simpler and more scalable, as it does not require manual or semi-supervised data acquisition.

4.3 Background

As detailed in Section 2.4 common self-supervised pretraining strategies include denoising tasks, which corrupt the original input by masking words (Devlin et al. 2019), identifiers (Lachaux et al. 2021), or sentences (Raffel et al. 2020; Wang et al. 2021b). The following sections discuss the aforementioned strategies and their effect when applied to source code.

4.3.1 Masked Language Modeling

Masked Language Modeling (MLM) introduces noise to the input by replacing words with mask tokens. Even though the task has proven to be successful on natural language, MLM is rather ineffective when used on source code, as previously argued. This is visualized in Figures 4.1a and 4.1b, where it can be seen that (tokenized) source code contains many tokens that are easy to predict when hidden. For instance whitespace and syntax tokens, such as `MASK1`, `MASK4`, `MASK5`, `MASK6`, `MASK8` in the figure, are easy wins for the model. Also, to recover that `MASK2` must be called `n` is trivial, since no other variable is in scope.

4.3.2 Regular Span Masking

Regular span masking is a denoising sequence-to-sequence task for pretraining LMs. Raffel et al. (2020) proposed this method for NLP, and it is also used for code in various models (Phan et al. 2021; Ahmad et al. 2021; Wang et al. 2021b). In this task, instead of replacing single words or tokens, multiple relatively short spans (mean of three tokens each) are

```
MASK1 fib( MASK2 ):
    # Compute the n MASK3 Fibonacci number.
    if n <= 1 MASK4
MASK5     MASK6 n
    return ( MASK7 (n-1) + fib(n-2 MASK8
```

(a) Noised input for masked language modeling (MLM).

```
MASK1 def
MASK2 n
MASK3 -th
MASK4 :
MASK5 □□□□
MASK6 return
MASK7 fib
MASK8 ))
```

(b) Output for MLM.

```
def fib MASK1
  MASK2 -th Fibonacci number.
    if n <= 1:
MASK3 n
    return fib(n-1 MASK4 (n-2)
```

(c) Noised input for regular span masking.

```
MASK1 (n):
MASK2   # Compute the n
MASK3       return
MASK4 ) + fib
```

(d) Output for regular span masking.

Figure 4.1: Input/output examples for the *masked language modeling* and *regular span masking* tasks, as used in CodeBert and CodeT5, respectively. The tasks are ineffective for code since many tokens can be predicted without reasoning about the code. For instance, predicting that the function parameter (`MASK2`) must be called `n` in (a) is trivial since no other variable is in scope. Additionally, many masked tokens are syntactical elements that do not require an understanding of the code's semantics, but only its syntax. For example, with `MASK3` in (c), only spaces and a syntactical keyword are masked. This leads to a less challenging training.

replaced by *sentinel tokens* in the input sequence, with an overall corruption rate of p_{mask}.

$$p_{\text{mask}} \coloneqq \frac{\sum_{i=1}^{k} \text{length}(\boldsymbol{Y}_i)}{\text{length}(\boldsymbol{c})} \approx 15\%. \tag{4.1}$$

The decoder predicts the hidden spans, separated and preceded by their specific sentinel tokens, in a single output sequence:

$$\boldsymbol{y} = \texttt{MASK1 the first masked span MASK2 the second masked span EOS} \tag{4.2}$$

However, similar to MLM, when regular span masking is applied to source code, it does not always produce challenging examples. This is illustrated in Figures 4.1c and 4.1d, where the spans often consist of syntactical elements or whitespace, such as `MASK3` that masks only whitespace and the keyword `return`, but misses the returned variable `n`. Even though this task is more challenging than MLM, it could be even more challenging. For instance when the full return statement including the variable would be masked, the model would have to reason what to return.

```py
def MASK1(MASK2):
    # Compute the n-th Fibonacci number.
    if MASK2 <= 1:
        return MASK2
    return MASK1(MASK2 - 1) + MASK1(MASK2 - 2)
```

(a) Noised input.

```
MASK1 fib
MASK2 n
```

(b) Output.

Figure 4.2: Input/output example for the *identifier deobfuscation* task. Masking identifiers does not affect the code's functionality but makes it harder to comprehend. In order to predict the original variable names, the model must comprehend the code's semantics, which can be a challenging task. Note that in this example, the mask tokens are used multiple times for each occurrence of the identifier.

4.3.3 *Identifier Deobfuscation*

Identifier deobfuscation is a structural sequence-to-sequence pretraining task for encoder-decoder models, introduced by Lachaux et al. (2021). The goal of identifier deobfuscation, as shown in Figure 4.2, is to restore the original names of identifiers in obfuscated code. Identifiers include variable and function names, but not method calls. Lachaux et al. (2021) replaced 50% of variable, function, and class names—but not method names—with distinct special tokens (e.g., `VAR1`, `FUNC1`, `CLASS1`) in the order they appear in code and trained the model to predict the original names.

Unlike regular span masking, the sentinel tokens are used multiple times in the input sequence, once for each occurrence of an identifier. The output sequence is constructed in the same way as in regular span masking. It should be noted that due to the construction of the output sequence, identifiers predicted at the end of the output sequence can see the deobfuscated identifiers that appear earlier in the sequence.

4.4 Approach

The approach in this chapter is to pretrain a encoder-decoder transformer model on a large corpus of source code using multiple sequence-to-sequence tasks. As detailed in Section 2.3, the transformer model encodes an input sequence of tokens $\boldsymbol{x}$ to generate an output sequence of tokens $\boldsymbol{y}$. For self-supervised learning, both the input and output sequence are constructed from a code token sequence $\boldsymbol{c} = (c^{(1)}, \ldots, c^{(l)})$. The noised input $\boldsymbol{x}$ is derived by replacing one or more subsequences $\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_k$ of $\boldsymbol{c}$ with special mask tokens $x^{(mask_1)}, \ldots, x^{(mask_k)}$, also known as sentinel tokens (Raffel et al. 2020). For $m \in 1, \ldots, k$ the subsequences $\boldsymbol{Y}_m$ are defined as $\boldsymbol{Y}_m = (c^{(i_m)}, \ldots, c^{(j_m)})$, where $i_m \leq j_m < i_{m+1} \leq j_{m+1}$ for all $m \in \{1, \ldots, k-1\}$.

$$\begin{aligned}\boldsymbol{x} = (&x^{(lang)}, c^{(1)}, \ldots, c^{(i-1)}, x^{(mask_1)}, c^{(j+1)}, \ldots, \\ &c^{(i'-1)}, x^{(mask_k)}, c^{(j'+1)}, \ldots, c^{(l)})\end{aligned} \tag{4.3}$$

$x^{(lang)}$ is a special token that indicates the programming language of the code, similar to the `[CLS]` token in BERT, see Section 2.1.1. The output sequence $\boldsymbol{y}$ is then constructed by concatenating (denoted by $\oplus$) the masked subsequences with the mask tokens:

$$\begin{aligned}\boldsymbol{y} &= (x^{(mask_1)}) \oplus \boldsymbol{Y}_1 \oplus \ldots \oplus (x^{(mask_k)}) \oplus \boldsymbol{Y}_k \oplus (x^{(eos)}) \\ &= (x^{(mask_1)}, c^{(i)}, \ldots, c^{(j)}, \ldots, x^{(mask_k)}, c^{(i')}, \ldots, c^{(j')}, x^{(eos)})\end{aligned} \tag{4.4}$$

The model is subsequently trained to reconstruct the hidden subsequences in $\boldsymbol{y}$ from the noised input sequence $\boldsymbol{x}$. Formulating all pretraining task using this masking scheme allows training all models and tasks with the regular cross-entropy loss from Equation (2.16) using teacher forcing. With this formulation no specific weighting of separate losses has to be done, instead the amount of samples each task contributes to the training data determines its weight. Each pretraining task defines only the input and output sequences, by defining the selection of the subsequences to be masked.

4.4.1 Pretraining Tasks

The goal is to create challenging tasks that require the model to reason about the semantic aspects of the code in order to detect useful patterns. As shown in Figure 4.1, regular tasks like MLM employed by CodeBERT and span masking used in CodeT5 do not consistently satisfy the above criteria. This section introduces tge novel structural pretraining task tree-based span selection and extebds the identifier deobfuscation task. The approach consists of a structural training pipeline that mixes multiple pretraining tasks. As shown in Figure 4.3, input/output examples are constructed dynamically by probabilistically selecting one of the following tasks for each sample:

1. **Tree-based span selection**, a novel structural task, where random subtrees are selected and masked is used in 33% of the samples (see Figure 4.4a).

2. **Identifier deobfuscation**, a structural pretraining task, where identifiers in the code (e.g., variable names, method names, etc.) need to be predicted. This task is used in 33% of the samples (see Figure 4.2a).

3. **Large span masking**, in which a single but sufficiently large span is masked is used in 30% of the samples. Generating large pieces of code aims to improve code generation capabilities of the model's decoder.

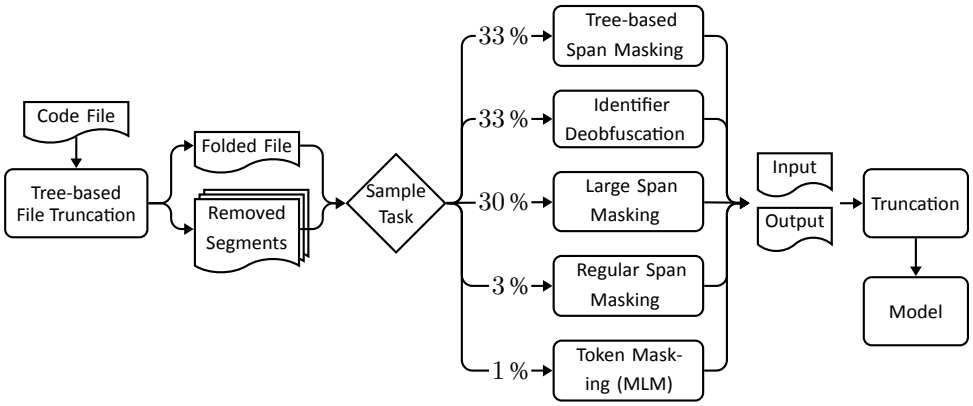


Figure 4.3: The training pipeline for the structural pretraining. First, complete code files are truncated into smaller samples by iteratively sampling subtrees and replacing the segment with a fold token (analogous to folding in an IDE). Both the folded file and all removed segments are used as individual samples to construct the input/output pairs according to one of the pretraining tasks. For each sample a pretraining tasks is sampled, which transforms it into a pair of a noisy input and the corresponding output sequence. In a final step, both sequences are truncated to a maximum length of 512 tokens before being fed into the model.

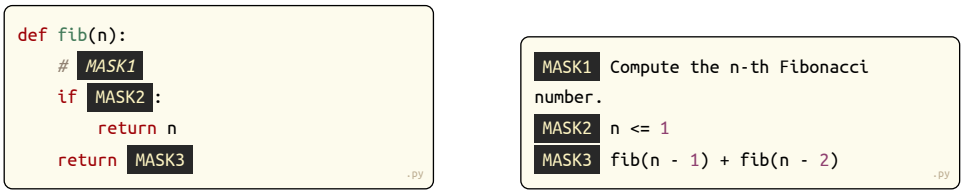
4. Finally, two regularization tasks enable the model to predict arbitrary spans at inference time: 3% of samples use the non-structural **regular span masking** task (see Figure 4.1c).

5. In 1% of the samples **masked language modeling** is used (see Figure 4.1a). This task is framed as a sequence-to-sequence task.

Please note that the pipeline is fully dynamic, and the model is exposed to new samples of various tasks at each epoch. First the code file is probabilistically truncated into smaller samples, then a pretraining task is drawn for each sample, which itself probabilistically constructs the input/output pairs. The tasks will be detailed in the following sections.

Tree-Based Span Selection

Tree-based span selection masks entire syntactic segments, such as comments, statements, or conditions. Figure 4.4 provides an illustrative example, where `MASK1` hides a method's docstring, which is a distinct node within the syntax tree. This produces without any supervision a setting similar to common evaluation tasks code summarization or code search. Another difficult challenge is achieved by `MASK2` that masks the condition of an `IfStatement`. Predicting the hidden segment requires analyzing the code's functionality and context. This way of masking leads to samples that more often require a deeper understanding of the code's functionality compared to masking individual regular spans. Additionally, tree-based span selection not only presents more challenging examples but also more closely mirrors a developer's workflow, for example when fixing a bug or adding new features.



(a) Noised input. **(b)** Output.

Figure 4.4: Input/output example for the **tree-based span selection** task. Here the masked spans are complete syntactic segments, producing challenging examples by design. For example, selecting the docstring of the method (`MASK1`) creates a task similar to code summarization and code search, two common code comprehension tasks.

To implement tree-based span selection, subtrees from the syntax tree are sampled and the corresponding code regions masked. This ensures that the targets represent syntactically complete code fragments. However, masking singular subtrees has a drawback: Adjacent statements or siblings in the syntax tree, such as multiple statements in a method body (e.g., the `IfStatement` and `ReturnStatement` in Figure 2.3), can't be masked together, as they do not share the same subtree and selecting the parent unit would mask the entire method body. To enable the combined masking of multiple, yet not all adjacent subtrees in the parent unit, the selection process may include direct siblings of the initially selected subtree where feasible.

Following Raffel et al. (2020) and Lewis et al. (2020b) multiple short subtrees are sampled until the corruption rate of approximately $p_{\text{mask}} \approx 15\%$ is reached, as in Equation (4.1). This work adopts sampling the individual span lengths from a Poisson distributions as in Lewis et al. (2020b). However, we use slightly longer spans than both of the aforementioned works of mean 8 tokens instead of 3, to create more realistic examples (such as the aforementioned comment). Formally, $\text{length}(Y_i) \sim \text{Pois}(8)$.

Identifier Deobfuscation

Building upon the work of Lachaux et al. (2021) described in Section 4.3.3, this thesis proposes some key refinements to the identifier deobfuscation task, which are discussed below. Let $\mathbb{I}$ be the set of all identifiers in the code $\boldsymbol{c}$, including method calls. To provide context for the model, a random percentage $p_{obf} \sim \mathcal{N}(60, 25^2)$ of the identifiers in $\mathbb{I}$ are selected for masking and a subset of the identifiers (denoted as $\mathbb{I}_{\text{mask}}$) is drawn from $\mathbb{I}$, so that $|\mathbb{I}_{\text{mask}}| = \text{round}(p_{obf} \cdot |\mathbb{I}|)$. The order of the identifiers in $\mathbb{I}_{\text{mask}}$ is shuffled, and the identifiers are then masked in the order in which they appear in the shuffled sequence. To mask identifier $I_i \in \mathbb{I}_{\text{mask}}$ each of its occurrences in $\boldsymbol{c}$ is replaced by the mask token $x^{(mask_i)}$ with a probability of 95%. Note that here $\boldsymbol{Y}_i = I_i$ so that in the output sequence $\boldsymbol{y}$, the identifier I_i is only predicted once, even if it occurs multiple times in the input sequence $\boldsymbol{x}$. This thesis does not use distinct mask tokens for different types of identifiers

(e.g., `VAR1` for variable names and `FUNC1` for method names). Instead, all identifiers are replaced with the shared mask tokens, i.e., `MASK1`, `MASK2`, etc.

Some of the design choices outlined above differ from the approach taken by Lachaux et al. (2021). The authors use a fixed percentage of masked identifiers and do not mention if the samples are pre-computed or constructed dynamically during training. Randomly selecting a percentage of identifiers to mask requires the model to adapt to varying levels of obfuscation across different samples (which we will do in Chapter 7). It introduces complexity and variability that is expected to more accurately reflects real-world coding practices. This approach allows using the model in different scenarios, where the level of obfuscation can vary. Additionally, extending masking to all types of identifiers, including method calls, is a notable improvement over prior work. Method calls shape the functionality of a piece of code. Predicting them not only increases the complexity of the task but also its practical relevance. Another design choice is the partial masking of identifiers, where each occurrence of an identifier within the code is masked with a probability of 95%. This approach can result in rare visible instances of an obfuscated identifier. We hypothesize that this allows the model to consider contextual clues for identification and reconstruction, but without overly relying on it in a blanket fashion (compare dropout). Moreover, the decision to randomize the order of masked identifiers in the input sequence differs from the method proposed by Lachaux et al. (2021), who masked identifiers in the order they appeared in the code. When the decoder predicts a specific identifier I_i, it can refer to deobfuscated identifiers $I_1, \ldots, I_{i-1}$ that appeared earlier in the output sequence. However, any subsequent identifiers $I_{i+1}, \ldots, I_k$ remain hidden. Randomizing the order aims to prevent the model from relying on the positional order of identifiers.

Large Span Masking

With the aforementioned structural pretraining tasks identifier deobfuscation and tree-based span selection, the model is mostly tasked to generate short sequences. This is a discrepancy to many fine-tuning tasks, such as code translation or bug fixing, that require the model to generate longer pieces of code. CodeT5 approaches this problem by using bimodal supervised data (pairs of code and natural language descriptions from the CodeSearchNet dataset) in addition to unsupervised pretraining (Wang et al. 2021b). In a second stage, the authors train the model to translate between code and natural language and vice versa. This trains the decoder to generate longer sequences of code, but has the drawback that it requires supervised data.

In this work, supervised data is avoided in favor of a purely self-supervised approach. To train the model to generate longer sequences, we employ the *large span masking* task. Here, only a single span Y_1 is masked, whereas length$(Y_1) \sim \mathcal{N}(50, 20^2)$, which masks approximately 10% of the code given that the maximum length of the input sequence is

512 tokens. Large span masking does not share the same pitfalls as regular MLM and short span masking, since it will always contain challenging code, given that the span is sufficiently large.

Regular Span Masking and Masked Language Modeling

In a small percentage of samples (<4%), one of the regular pretraining tasks *regular span masking* (<3%) or *masked language modeling* (<1%) is used. Although these tasks may not always present strong challenges, they enable the LM to process arbitrary spans during inference. Therefore, these tasks are mixed in at a low percentage of samples to ensure the model learns to predict all types of gaps.

The implementation of regular span masking closely follows the implementation in the original T5 model (Raffel et al. 2020). Span lengths are sampled from a Poisson distribution, whereas $\text{length}(Y_i) \sim \text{Pois}(4)$. For masked language modeling individual tokens are masked, but the output sequence is constructed in the same way. The total corruption rate by these two tasks is approximately 15%.

4.4.2 Tree-based File Truncation

During the training phase, both the $\boldsymbol{x}$ and $\boldsymbol{y}$ are constrained to a maximum of 512 tokens[3]. In the pretraining dataset, the average number of tokens per file is approximately 1890. Consequently, it becomes necessary to truncate large code files into smaller units. Truncation for code is usually approached by one of the following strategies:

1. Files are segmented into blocks of code. This method, however, results in the code at the end of the file being isolated from the code at the beginning. Both segments are never seen together, which is suboptimal for training.

2. Many approaches use method-level segments, which even stronger restricts the visibility of the full context (Wang et al. 2021b).

However, truncating code files in a way that allows the model to reason over inter-file contexts could be beneficial. For instance, in the context of software development, understanding a class' signature and its method definitions is often enough to get an overview of the code's functionality, even without full access to the method contents. To address these limitations, this thesis proposes an alternative truncation strategy, called *tree-based file truncation*, that utilizes tree-based span selection (see Section 4.4.1). It is inspired by code folding in an Integrated Development Environment (IDE), which most software developers use to navigate large code files by focussing on specific parts and hiding segments such as comments or method contents.

[3] The T5 model is capable of encoding and generating sequences of arbitrary length at inference time, due to its bucketed relative positional embeddings (refer to Section 2.3.2).

With *tree-based file truncation* large code files are truncated by randomly collapsing large code segments. Such a collapsed region could be the content of a method, so that only the definition remains, or multiple methods altogether. As shown in the training pipeline in Figure 4.3, not only the folded code file is used as a sample, but also the removed segments are treated as individual samples. This method produces short to medium-sized pieces of code with varying levels of context. It is implemented using tree-based span selection, where instead of sampling small segments for masking, larger segments of 150 to 800 tokens are randomly selected. Here, subtrees are sampled proportionally to their size, so that larger segments are more likely to be selected. All removed segments are replaced with the same `[FOLD]` token, which signals the model that some code is missing at this position, but it does not need to be predicted.

Note that this is an initial step in the training pipeline shown in Figure 4.3 that produces smaller samples $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots$ from the full code file $\boldsymbol{c}$. Each sample is then used individually to construct the input/output pairs according to one of the pretraining tasks. In this step segments up to 800 tokens are selected, even though the final input and output sequences are limited to 512 tokens. However, each task further shortens the segment by masking parts of it. At the end, if either of the input and output example remains longer than 512 tokens, it is as a final step truncated to the maximum length on a token basis.

4.5 Experimental Setup

The goal of the experiments conducted in this chapter is to evaluate the effectiveness of the structural pretraining tasks introduced in the previous section. The overarching research problem is whether the structural tasks provide a more effective way to pretrain code understanding models than regular pretraining tasks. To this end, SyntaxPT is compared against a baseline trained with regular pretraining tasks on the same dataset and setup. This ensures that the only variable between the models is the pretraining tasks. To accurately measure the code understanding capabilities of the models, the performance is evaluated on benchmark datasets that aim to measure code understanding capabilities.

Code understanding encompasses a broad spectrum of application scenarios, including code summarization, generation, translation, retrieval, and bug detection. All these applications require the model to reason about the code's functionality, structure, and semantics. The CodeXGlue benchmark proposed by Lu et al. (2021) is a collection of such tasks and accompanying datasets specifically designed to accurately evaluate the performance of code LMs across a wide range of code understanding tasks. It has been adopted as the evaluation benchmark for this work, as it provides a fair comparison to the state-of-the-art models[4]. In this work, five evaluation tasks from the CodeXGlue collection are selected.

[4]To clarify, the evaluation benchmark datasets used in this chapter differ from those used in Chapter 3 because the CodeXGlue benchmark is a more recent, difficult, and comprehensive collection of code

To assess the performance of the models, each model is fine-tuned on the task's respective training data and subsequently evaluated according to the task's evaluation metric. Since CodeXGlue provides evaluation scripts for each task, this ensures a fair comparison between the models and the state-of-the-art.

In addition to the comparison of the structural model to the baseline model which is trained only on regular pretraining tasks, the benefit of pretraining is also evaluated by comparing the models to an end-to-end model. The end-to-end model is trained without any pretraining, directly on the fine-tuning tasks. It allows not only to measure the overall benefit of pretraining by comparing it to the pretrained models, but also to measure the effectiveness of the proposed preprocessing pipeline, since it is compared to other recently proposed models, that are also trained end-to-end. To summarize, in the experiments of this chapter three models are trained and evaluated on the CodeXGlue benchmark:

1. A model which is trained **end-to-end** on each task, without any pretraining. This model is referred to as Transformer (no PT).
2. The **baseline model** is trained with regular pretraining tasks, excluding the structural tasks. This model is referred to as RegularPT.
3. The **structural model** is trained with the structural pretraining pipeline introduced in this chapter. This model is referred to as SyntaxPT.

The experiments demonstrate that the structural model outperforms both baseline models and also many state-of-the-art models on most tasks and that structural pretraining leads to a much more consistent fine-tuning performance across all tasks.

This section details the experimental setup, including the model architecture, the regular pretraining tasks for the baseline model, the tokenizer, the pretraining dataset, implementation details, and the evaluation benchmarks.

4.5.1 Research Questions

The aforementioned experimental setup aims to answer the following research questions:

Research Question 4.1: *Does structural pretraining provide a stronger learning signal than regular pretraining, with respect to code understanding capabilities?*

This question investigates whether incorporating structural pretraining tasks enhances the model's ability to understand code more effectively than traditional pretraining methods. To this end, the structurally pretrained model (SyntaxPT) is compared to the model pretrained on regular tasks (RegularPT) on the CodeXGlue benchmark, as outlined above.

understanding tasks. However, one task, the code summarization task from the CodeSearchNet dataset, remains the same as in Chapter 3.

Research Question 4.2: *What is the benefit of pretraining on code compared to training from scratch?*

This question aims to measure the advantages of pretraining for the code understanding capabilities of the model. To this end, the non-pretrained model (Transformer (no PT)) with both the regular pretrained model (RegularPT) and the structurally pretrained model (SyntaxPT) on the CodeXGlue benchmark.

Research Question 4.3: *How does the structural pretraining compare to the Rst model from Chapter 3?*

This question aims to assess the effectiveness of the structural pretraining approach in comparison to the structural transformer model of Chapter 3. Both models are evaluated on code summarization task from the CodeSearchNet dataset (which is part of the CodeXGlue benchmark).

Research Question 4.4: *How do the proposed models perform compared to other state-of-the-art models?*

This question evaluates the performance of the proposed models against existing state-of-the-art code-LMs. This will provide insights about the effectiveness of the preprocessing pipeline and the structural pretraining tasks.

4.5.2 Model Architecture

Instead of using the original transformer architecture proposed by Vaswani et al. (2017), like in Chapter 3, the more recent T5 architecture from Raffel et al. (2020) is used (see Section 2.3 for a description of the transformer architecture). The T5 architecture contains various small improvements over the original transformer model. Specifically, Raffel et al. (2020) remove the layer norm bias, place the layer normalization outside the residual path, and use relative positional embeddings instead of absolute (see Section 2.3.2).

In particular, this chapter uses the `T5-base` configuration, in which both the encoder and decoder sub-models consist of 12 layers, with a hidden size of $d = 768$ and 12 attention heads operating on key and value vectors of dimensionality $d_h = 64$. The feed-forward layer has a dimensionality of $d_{\text{ff}} = 2048$. The relative positional embeddings do not learn a separate embedding for each possible relative position—as the model in Chapter 3—but instead use 32 logarithmically increasing buckets for the positions (compare Section 2.3.2). A dropout rate of 0.1 is applied throughout the model, and word embeddings are not shared between the encoder and decoder. Differing to the original T5 model, the Gated Gaussian Error Linear Unit (GEGLU) activation function (Shazeer 2020) that is used in newer versions of the model is also used in this chapter. The model used in our experiments has the same architecture as CodeT5. This chapter trains a custom BPE tokenizer so that the vocabulary of the model contains 32k tokens (described in Section 4.5.4), and

the model has approximately 247 million parameters. This architecture with the same tokenizer is used for all models in the experiments.

4.5.3 Baseline

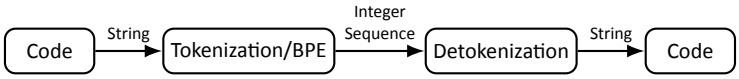
To evaluate the impact of the structural pretraining tasks on the model's performance, the model is compared to a baseline model trained exclusively with regular pretraining tasks. For a fair comparison, the baseline model is kept as similar as possible to the structural model. Only the structural pretraining tasks—identifier deobfuscation and tree-based span selection—are replaced by utilizing regular pretraining tasks with higher frequency: token span masking and large span masking. The baseline pretraining tasks are configured as follows:

1. **Regular span masking** is applied to 49% instead of 33% of the training samples. In this task, spans of length length$(Y_i) \sim \text{Pois}(8)$ are masked, with a total corruption rate of 15%. This configuration is similar to that in the structural model for the tree-based span selection task.
2. **Large span masking** masks a single, sufficiently large span and is also used in the structural model. In the baseline it is applied to 47% of the training data (opposed to 30%), with the same configuration as in the structural model.
3. The structural model uses two regular pretraining tasks to ensure that the model can predict arbitrary spans at inference time. Both are kept in the baseline model. Hence, 3% of the samples use **regular span masking** with a span length of 4 tokens. Note that this task is used twice in the baseline with different sample lengths (8 and 4), to have a fair comparison.
4. In the remaining 1% of the samples, **masked language modeling** is used.

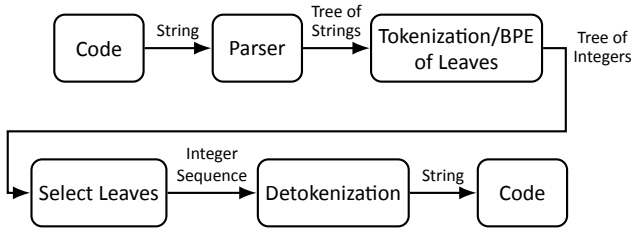
Please note that tasks (3) and (4) of the baseline model are exactly the same as tasks (4) and (5) of the structural model defined in Section 4.4.1.

4.5.4 Tokenizer

The transformer model operates on an input sequence of integers, which are obtained by tokenizing the input text. The tokenizer significantly influences model performance and the kind of possible outputs. A generative LM should be able to produce all types of code and text. The tokenization process can be seen as a function tokenize : $String \mapsto \mathbb{N}^n$ that maps a string (the code) to a sequence of integers. If tokenize is bijective the tokenization is reversible and the original input can be restored. This is shown in Figure 4.5a. In the last chapter relatively short natural language output sequences were generated. Thus, the tokenization process from Chapter 3—that tokenized code by splitting on camel case and subsequently applied BPE to the split identifiers—ignored whitespace. This makes a



(a) Regular-tokenization process.



(b) Tree-tokenization process.

Figure 4.5: Comparison of regular- and tree-tokenization. Both processes are reversible, but the tree tokenization process allows for direct structural manipulation of the tree in its tokenized form.

reversible detokenization impossible. Consider the example `foo(a, b)`, which would be tokenized into `foo`, `(`, `a`, `,`, `b`, `)`. The detokenizer can never recover the information that there existed a space before parameter `b`, i.e., this information is lost. However, in this chapter the LM should (1) be able to produce any kind of code/natural language sequences, with any type of formatting or whitespace. Thus, the tokenization process should be reversible.

In addition to a reversible tokenization the tokenized code should (2) be able to be stored as a tree of integers for fast and dynamic structural manipulations in the training pipeline, since the tokenization process that turns strings into integer sequences is an expensive operation. Therefore, common practice is to BPE tokenize the data and store it as integer sequences. This enables fast loading and training on large datasets. Constructing the training examples using one of the regular pretraining tasks, such as masking words or spans, is straightforward and done directly on the tokenized data. However, for the structural pretraining tasks it is necessary to have structural information, which is lost in this process. It can't be recovered from the tokenized data, since the BPE tokenizer may merge tokens that belong to separate syntactical structures into a single token (e.g., `if(`). First parsing and then tokenizing the leaves of the tree is the solution.

The tree tokenization process proposed in this chapter is shown in Figure 4.5b. It allows for direct structural manipulation of code in its tokenized form as a tree of integers. It starts with parsing code into a Concrete Syntax Tree (CST) using the tree-sitter syntax parser (Brunsfeld 2023). Thereby, tree-sitter is modified so that the syntax tree also includes whitespace, which it usually discards. An example Concrete Syntax Tree (CST) for the Fibonacci function is shown in Figure 2.3. It arranges all code and whitespace tokens in a

tree structure, so that the leaves are the actual tokens that appear in the code. *Concatenating all leaves gives back the original code file.*

To transform the CST—which still contains the tokens as strings—into a tree of integers, first all non-terminal nodes are added to the vocabulary. These can be directly encoded. However, expressive terminal nodes—such as identifiers, or even the complete docstring of a method—need to be further tokenized, for which BPE is used. Each leaf is encoded separately, which splits the leaf-node into multiple tokens. To arrange the split tokens in the tree, a new node `[BPE]` is introduced. The node replaces the leaf-node and the split tokens are arranged as children of the artificial node. This preserves the property of a CST, in which the code can be reconstructed from the leaves of the tree, as mentioned above. Note however, that this method differs slightly from the approach described in Section 3.4.2, where the new nodes were arranged in a sequential chain. With this tree of integers, many manipulations such as replacing identifiers can be done easily, and at the same time a tokenized form of the original code can always be obtained by selecting the leaves.

Now the question remains whether an existing tokenizer should be reused or if it is better to have a separate tokenizer more tailored on the unique characteristics of source code. For example, CodeBert uses the reversible RoBERTa tokenizer (Liu et al. 2019). However, this tokenizer is made for natural language, and using it for code has a few drawbacks:

- It uses a large vocabulary of 50k tokens, many of which are rarely found in pre-tokenized code. For example a lot of tokens start with whitespace, e.g., `" return"`. Such a token is rare in a CST, in which whitespace is a separate leaf in the syntax tree. It only appears in comments or strings.

- Whitespace is handled expressively. Each space is a separate token, which is not optimal for code in which whitespace is used to format the code.

In order to approach these issues and keep the size of the vocabulary at a minimum, while still using the more expressive tree-tokenization approach, a custom BPE model is trained on files from 16 different programming languages. Following the line of research of Liu et al. (2019) a byte-level base vocabulary is used, which allows for the encoding of every Unicode character from the byte-level units (Radford et al. 2019). Based on these 256 base units, 30k merges, or subwords, are learned, which keeps the vocabulary and the model size at a manageable dimensionality. Additionally, space for 100 special tokens, such as language identifier and mask tokens, is reserved. Furthermore, given that code consists of significant amounts of whitespace, specific whitespace tokens (e.g., 4 spaces, 8 spaces) are added to the tokenizer. This effectively reduces the total number of tokens in the tokenized code compared to the RoBERTa tokenizer. In total the vocabulary of the

tokenizer consists of 31,914 tokens, which includes all nonterminal nodes of the CST for all languages (see Table 2.1), the subword-tokens, and the added special tokens.

The Tensortree Library

Neural networks, their training routine, but also data processing are implemented using PyTorch (Paszke et al. 2019) or TensorFlow (Abadi et al. 2016), which provide fast and effective tensor operations, and automatic differentiation. This thesis focuses on PyTorch, which is used for all experiments. In this chapter the training pipeline consists of code that is initially transformed into a tree structure of integers, as shown in Figure 4.5b. This structure is then used to generate input and output examples based on the specific pretraining task. For instance, in the identifier deobfuscation task, variables need to be identified within the syntax tree and subsequently replaced with mask tokens. After manipulation, the leaves of the tree are concatenated to form the input vector $\boldsymbol{x}$ for the transformer model.

To enable the fast processing of trees of integers within the PyTorch framework, the tensortree library was developed in this thesis (**Villmow** 2021). The library stores trees in PyTorch tensors, which allows the user to use common PyTorch operations on the tree. Specifically, the tree is flattened and stored as a pre-order node sequence in a tensor, while the tree's structure is maintained in two separate distinct arrays. In these arrays, tensortree stores for each node

1. the index of the parent node, and
2. the number of descendants.

It should be noted that these structural representations can be converted into one another in $O(n)$ time, where n is the number of nodes in the tree. Storing an array of parent pointers is a common practice, but also having pre-computed access to the number of descendants for tensor-based tree-manipulations is, to the author's knowledge, a novel approach in the deep learning domain. Even though two structural representations require storage of $3n$ integers, the benefits of having both representations outweigh the additional storage cost. The library also provides functions to convert between the two representations, as well as to sample subtrees, and to identify nodes based on their properties. These operations make use of PyTorch's tensor operations, which are optimized for fast and efficient computation:

- `tree.nodes[tree.descendants == 0]` to create a mask for all leaf nodes and subsequently select them to form the input sequence $\boldsymbol{x}$,
- `tree.nodes[tree.parents[idx]]` to access the data of a specific node's parent,

- `tree.nodes[idx: idx + tree.descendants[idx] + 1]` to extract the subtree at index `idx`,
- `(tree.nodes == identifier).nonzero()` computes the indices of identifier nodes, assuming that `identifier` represents the id of an identifier node.

Additionally, the library can be used to compute characteristic tree matrices such as the node incidence matrix from Equation (3.2), to identify nodes with PyTorch native comparison and mask operations, as demonstrated above, and to sample subtrees by its size using the descendants array. However, TENSORTREE has been designed to efficiently analyze, identify, and select nodes, but not to iteratively modify or build trees. Modifications to the tree structure are resource-intensive, as the three tensors must be recomputed and reallocated. Nevertheless, this is a common use case and with performance in mind, tree modifications like insertion, deletion, and replacement are still supported in TENSORTREE. To do so, bulk operations can be used that improve performance over iterative operations. A typical workflow would be to first identify the indices of identifier nodes by computing a mask over the nodes, as shown above. Once the identifier indices have been determined, a bulk operation can be employed to replace every subtree simultaneously, resulting in a new tree with the identifiers replaced. Exemplary code for this is provided in Figure A.3 in the appendix. The TENSORTREE library has been made available as an open-source project on PyPi (**Villmow** 2021).

4.5.5 *Pretraining Dataset*

Language	Tokens	Files	Tokens per File
C	6.5B	2.2M	2904
C#	4.2B	2.8M	1479
C++	8.7B	3.7M	2317
CSS	0.5B	0.4M	1498
Go	4.1B	1.8M	2351
Haskell	0.2B	0.1M	1854
Java	12.5B	7.3M	1702
JavaScript	6.9B	4.5M	1550
Julia	0.04B	0.03M	1175
OCaml	0.1B	0.06M	2200
PHP	3.4B	2.2M	1550
Python	6.3B	3.0M	2079
Ruby	1.1B	1.1M	1046
Rust	0.9B	0.4M	2419
Scala	0.3B	0.3M	1123
TypeScript	2.5B	2.3M	1091
Total/Average	58B	32M	1890

Table 4.1: The dataset used for pretraining the model. It consists of code from 237k repositories sourced from GitHub. The table shows the number of tokens (in billions), files (in millions), and the average number of tokens per file for each programming language.

GitHub is the largest platform for hosting code repositories in the current software development landscape. In this work it is used to collect a large and diverse dataset of open source code for pretraining the model. It should be noted that not all repositories that can be found on GitHub are suitable for pretraining, as GitHub also contains personal projects, low-quality code, or code that is not actively developed. In this thesis two criteria

are used to select high-quality repositories for the pretraining dataset: the number of GitHub stars and the activity of the repository (Borges and Valente 2018).

With GitHub stars users can favor repositories and many use it to bookmark repositories. The amount of stars a repository receives can be used as a proxy for its popularity. Given the assumption that popular repositories are more likely to contain high-quality code, the amount of stars a repository has is used as a criterion to select repositories for the pretraining dataset. Another measure for the quality of a software project is the activity of its repository. Since open source software development is a collaborative process, developers contribute to a project by creating pull requests that resolve issues or add features. Therefore, repositories with active pull requests are more likely to be actively developed, maintained, and of higher quality. These two criteria are employed to include only repositories with

1. more than 10 GitHub stars,
2. active development, measured by having a pull request between April and September 2021.

It is not possible to identify repositories that meet these criteria using the free GitHub API, because of the API's low rate limits. However, an alternative is the Google BigQuery dataset *GitHub Activity Data* (GitHub Inc. 2024), which contains a monthly snapshot of all public events on GitHub. It was used in this work to identify active repositories with more than 10 stars[5]. After cloning the repositories, only files from programming languages for which a tree-sitter parser is available were kept. On GitHub, forking is a common practice whereby developers copy a repository to their account and make changes to it, for example to fix bugs or add features. Many forks become popular and are starred by users, which can lead to numerous duplicate files in the dataset. To account for forks, duplicates were removed on a file level.

Ultimately, the pretraining dataset consists of 32 million files in 16 programming languages from 237k repositories, with a total of 58 billion tokens after tokenization. Precise statistics of the dataset for each programming language are presented in Table 4.1. A subset of 1000 repositories comprising 100k files was reserved for validation and testing. Thereby, a language distribution of the validation set similar to that of the training data was maintained.

4.5.6 Pretraining Hyperparameters and Setup

Optimizing the throughput is an important aspect when training LMs on large datasets. To maximize the amount of tokens each GPU processes in a batch, the batch size is dynamically adjusted, so that a maximum of 6000 input tokens are processed in a batch. To do so, samples are grouped by their size, and batches are created with samples of similar

[5]The query used to collect the information is shown in Figure A.2 in the Appendix.

sizes to minimize padding. On average and without padding tokens, each batch contained approximately 5700 input tokens and 1100 target tokens.

For optimization, the AdamW optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$, and $\epsilon = 1e{-}6$ was used with a peak learning rate of $2e{-}4$. Following Liu et al. (2019), the learning rate was warmed-up over the first 6% of the total training steps, which is 60,000 steps. After the warm-up phase, the learning rate was annealed to 0 with a polynomial decay. All experiments were conducted on a DGX system equipped with 8 A100 GPUs, each with 40GB of memory. Mixed precision training with BFloat16 was used to reduce the memory footprint and to increase the training speed. To further increase the batch size, which has been shown to be effective in LM training (Liu et al. 2019), and mimic training with 64 GPUs, gradients were accumulated over 8 steps, which results in an effective batch size of approximately 384,000 tokens[6]. The models were implemented using PyTorch (Paszke et al. 2019), making use of the PyTorch Lightning (Falcon and Team 2019) library for multi-GPU and mixed-precision training.

The models wetr trained for a total of 1 million steps, and each training takes approximately 3 weeks using the above hardware and configuration. A training processes around 365 billion input tokens, and predicts approximately 70.4 billion tokens. Because of the long training duration, extensive hyperparameter tuning was not feasible. The hyperparameters were instead chosen based on previous work and the available computational resources (Raffel et al. 2020; Wang et al. 2021b).

4.5.7 Fine-Tuning Tasks and Datasets

After pretraining the model it is fine-tuned on an evaluation task from CodeXGlue to assess its code understanding capabilities. In this chapter five tasks from the CodeXGlue benchmark (Lu et al. 2021) were selected for evaluation. CodeXGlue provides a specific version of each dataset and additionally an evaluation script for every task, so that a fair comparison with other state-of-the-art models is possible. Each task comes with predefined training, validation, and test splits. Table 4.2 shows the number of samples in each split for each task and dataset. Metrics are reported using the provided evaluation scripts. The tasks include three generation, one classification, and one retrieval task:

1. **Code Translation**: Translates code snippets between programming languages (generation).

2. **Code Refinement**: Removes bugs from code snippets (generation).

[6]The authors of the RoBERTa model reported best results for an effective batch size of 1M tokens, but achieved similar performance with 130k tokens (Liu et al. 2019). For performance reasons a setting in between the two was chosen in this work.

Task	Type	Source	Name	Subset	Train	Valid	Test
Code summarization	Generation	Husain et al. (2019)	CodeSearchNet	Ruby	24,927	1,400	1,261
				JavaScript	58,025	3,885	3,291
				Go	167,288	7,325	8,122
				Python	251,820	13,914	14,918
				Java	164,923	5,183	10,955
				PHP	241,241	12,982	14,014
Code translation	Generation	Lu et al. (2021)	-	-	10,295	499	1,000
Code refinement	Generation	Tufano et al. (2019)	-	Small	46,680	5,835	5,835
				Medium	52,364	6,546	6,545
Defect detection	Classification	Zhou et al. (2019)	Devign	-	21,854	2,732	2,732
Clone detection	Retrieval	Mou et al. (2016)	Poj-104	-	32,000	8,000	12,000

Table 4.2: Number of samples in the training, validation, and test set for each task from the CodeXGlue benchmark.

3. **Code Summarization**: Generates natural language descriptions for given code snippets (generation).
4. **Defect Detection**: Detects vulnerabilities in the code snippets (classification).
5. **Clone Detection**: Retrieves similar code snippets that implement the same functionality as the given one (retrieval).

Fine-Tuning Hyperparameters and Setup

As previously detailed three models are trained and evaluated in the experiments: (1) a non-pretrained model that is trained end-to-end on the task, (2) the baseline model trained only with regular pretraining tasks, and (3) the proposed structural model. Each model is trained on the training dataset of the specific task. A separate hyperparameter sweep is conducted for every model and the performance of each model on the validation set is optimized. At the end of the sweep only the best performing run of every model is evaluated once on the test set using the provided evaluation scripts. Generation tasks, such as code translation and summarization, follow the same sequence-to-sequence training paradigm described in Section 4.4. Thus, to fine-tune the model for generation tasks only the input and output sequences need to be changed to the specific task. The classification and retrieval tasks require architectural changes. These are described in the respective task sections.

As for pretraining, the task-specific fine-tuning is implemented with PyTorch (Paszke et al. 2019), PyTorch Lightning (Falcon and Team 2019), and the models are trained with mixed precision and a dropout rate of 0.1. Some general hyperparameters slightly differ from the pretraining setup. For example for fine-tuning typically a smaller learning rate is used, to avoid a phenomenon called catastrophic forgetting in which models forget previous knowledge (Goodfellow et al. 2014, p. 194). Furthermore, since the focus is less on

maximizing throughput, the batch size h_b is fixed during fine-tuning and not dynamically adjusted, as it was during pretraining. Similarly, the AdamW optimizer (Loshchilov and Hutter 2019) is used with a learning rate schedule, in which the learning rate h_{lr} is warmed up over the first 10% of the training steps and then decayed linearly until the maximum number of training epochs is reached. Note that with this setup, the maximum amount of training epochs h_e influences the learning rate schedule.

These hyperparameters can have a strong influence on the model's performance, especially on the CodeXGlue tasks, and thus to find the optimal set the Bayesian hyperparameter optimization strategy from Weights and Biases (Biewald 2024) was used. The specific values and ranges that have been swept over are mentioned in the respective task sections. Early stopping is used, as described in Section 2.2.1, storing the best checkpoint of that run, and stopping the run when performance does not improve over three consecutive epochs. At the end of the sweep, only the best checkpoint according to the validation performance across all runs is evaluated once on the test set. On generation tasks, the predictions were generated greedily during validation, while at test-time a beam search with beam size of 5 was used.

To construct input examples for the fine-tuning tasks, the same tree tokenization process as in pretraining is used, and the leaves of the tree are concatenated to form the input or output sequence. As in pretraining, a language identifier token is prepended to the input sequence.

Code Translation

Code translation is a sequence-to-sequence task that aims to translate code snippets from one programming language to another. In the code translation dataset from CodeXGlue, the task is to translate code snippets from Java to C# and vice versa (Lu et al. 2021). As shown in Figure 4.6 the code snippets are rather small, contain only a few lines of code, in which mostly naming conventions and syntax differ between the two languages.

```cs
public int GetCellsVal(){
  int size = 0;
  foreach (char c in cells.Keys){
    Cell e = At(c);
    if (e.cmd >= 0){size++;}
  }
  return size;
}
```

(a) Code snippet in C#.

```java
public int getCellsVal() {
  Iterator<Character> i = cells.keySet().iterator();
  int size = 0;
  for (; i.hasNext();){
    Character c = i.next();
    Cell e = at(c);
    if (e.cmd >= 0){size++;}
  }
  return size;
}
```

(b) Code snippet in Java.

Figure 4.6: Example of a code snippet in C# and Java from the code translation dataset (Lu et al. 2021).

Because the dataset contains relatively few samples—details are shown in Table 4.2—a good understanding of the programming language is required to perform well on this task. The models are evaluated using the Exact Match (EM) metric (which is the same as accuracy). It measures the percentage of code snippets in which the complete translation has been generated correctly. Additionally, the smoothed BLEU score is reported.

Hyperparameters In the sweep the fine-tuning batch size h_b is drawn from the set $\{8, 12, 16, 24, 32\}$, the learning rate h_{lr} is drawn from the interval $[10^{-5}, 10^{-4}]$, and the maximum number of epochs h_e is drawn from $\{5, 8, 10, 15\}$ for the pretrained variants and $\{10, 15, 20, 25\}$ for the non-pretrained model to allow for more training. To combat the small dataset size, code snippets are translated in both directions at the same time during training, i.e., from Java to C# and from C# to Java. During validation and testing each direction is evaluated separately, but the sweep optimizes the average accuracy of both directions. Each pretrained run takes around 1.5h on a single A6000 GPU, so that in a 7 days compute limit around 150 runs per model are conducted. The non-pretrained model has the same compute limit, but had to be trained for more epochs, so that only around 50 runs have been conducted.

Code Refinement

Code refinement is a sequence-to-sequence task that aims to automatically apply fixes to a piece of Java code as shown in Figure 4.7. The fixes can be small bug fixes, such as removing unused variables or statements, missing arguments, or more complex changes, such as adding null checks. The dataset has been introduced by Tufano et al. (2019) and is part of the CodeXGlue benchmark (Lu et al. 2021). The authors created the dataset by collecting commits from open-source projects that had keywords that indicated a bug fix in their commit message (e.g., *fix*, *bug*, *error*, or *issue*). The authors fetched the code snippets before and after the bug fix and subsequently extracted method-level pairs of buggy and fixed code. An abstraction process was applied to the code snippets, which included removing comments and annotations, normalizing identifiers such as method and variable names, and replacing literals with placeholders.

The authors diveded the dataset into two subsets, small and medium, based on the length of the code snippet and included a train, validation, and test set for each subset. During creation of the splits the authors used clone detection tools to avoid data leakage by verifing that the code snippets in the test set were not present in the training or validation set.

Hyperparameters The batch size h_b is drawn from the set $\{8, 16, 32, 64\}$, the learning rate h_{lr} is drawn from the interval $[5 \times 10^{-5}, 10^{-4}]$, and the maximum number of epochs h_e is drawn from $\{5, 8, 10, 15\}$ for the pretrained variants and $\{10, 15, 20, 25\}$

```java
private void METHOD_1( ){
    if((VAR_1.length) > 1 ){
        VAR_2 = (VAR_1.length) - 1;
        METHOD_2(VAR_1[VAR_2]);
    }
}
```

(a) Buggy code snippet with a missing null check.

```java
private void METHOD_1(){
    if((VAR_1) != null){
        if ((VAR_1.length) > 1 ){
            VAR_2 = (VAR_1.length) - 1;
            METHOD_2(VAR_1[VAR_2]);
        }
    }
}
```

(b) Corrected code snippet.

Figure 4.7: Example of a buggy and the fixed code snippet from the code refinement dataset (Tufano et al. 2019).

for the non-pretrained model to allow for more training. The model was trained jointly on both subsets of the dataset, but validation accuracy is computed separately for each subset. The sweep maximizes the average accuracy over both subsets. Each training takes approximately 14h on a single A6000 GPU, and a compute limit of 15 days is set. In that compute limit around 35 runs were conducted for all models.

Code Summarization

Code Summarization aims to generate a natural language description for a function. The CodeSearchNet dataset (Husain et al. 2019) is used for this task, which contains code snippets in six different programming languages: Ruby, JavaScript, Go, Python, Java, and PHP. The same task and dataset was used to evaluate the structural transformer model in Section 3.4.3. However, at the time of the experiments of the last chapter, it was not part of the CodeXGlue benchmark. A brief description of the task is given here for completeness. Consider the example in Figure 4.8, where a Python function needs to be summarized as `return reverse complement of read`.

```python
def rev_c(read):
    rc = []
    rc_nucs = {'A':'T', 'T':'A', 'G':'C', 'C':'G', 'N':'N'}
    for base in read:
        rc.extend(rc_nucs[base.upper()])
    return rc[::-1]
```

Figure 4.8: Python function that reverses and complements a read.

As shown in Table 4.2, the size of this dataset is relatively large. In total the combined training set contains over 900k samples. This dataset is an order of magnitude larger than the other evaluation benchmarks, which range from 10k to at most 100k training samples. Given the size of this dataset, many proposed code LMs, such as CodeT5, use

this bimodal dataset during pretraining. Additionally, the dataset is slightly imbalanced, with the majority of Python and PHP samples, closely followed by Go and Java.

Hyperparameters On this task the input sequence $\boldsymbol{x}$ is truncated to have at most 400 tokens and the output sequence $\boldsymbol{y}$ to 128 tokens. Given the large size of the dataset the batch size is not optimized in the hyperparamter sweep. Instead, a fixed batch size of 24 sequences is used, as is this is the largest batch size that fits on the GPUs. With the hyperparameter sweep, the learning rate h_{lr} is drawn from the interval $[10^{-5}, 10^{-4}]$, and the maximum number of epochs h_e is drawn from $\{5, 10\}$. The models are trained on the combined training set of all programming languages, but the validation and test accuracy is computed separately for each language. The sweep optimizes the average of the smoothed BLEU scores over all languages. Each training takes approximately 12h-24h on three A6000 GPUs, and a compute limit of 22 days is set. In that compute limit around 30 runs were conducted for all models.

Defect Detection

The defect detection task aims to identify vulnerabilities, such as memory leaks or use-after-free vulnerabilities in C functions. The devign dataset—that is part of the CodeXGlue benchmark (Lu et al. 2021)—has been introduced by Zhou et al. (2019) and is used here. The authors manually annotated vulnerability fixing commits from large open-source projects, such as the Linux kernel and extracted functions with labels.

Model Defect detection is a binary classification task where the model predicts whether a given code snippet contains a defect or not. Differing to CodeT5, this thesis does not treat this task as a sequence-to-sequence task, where the decoder generates the class label. Instead, only the encoder part of the model is used, and a classification head predicts the binary label. Formally, the hidden state of the language identifier token of the encoder's last layer $\boldsymbol{h} \in \mathbb{R}^d$ is fed into a classification head, to predict the logits for the binary label $\hat{y} \in \{0, 1\}$ (see Section 2.2.2):

$$\boldsymbol{r} = \tanh(\boldsymbol{W}_1 \,\text{dropout}(\boldsymbol{h}) + \boldsymbol{b}_1) \tag{4.5}$$

$$\hat{\boldsymbol{y}} = \boldsymbol{W}_2 \,\text{dropout}(\boldsymbol{r}) + \boldsymbol{b}_2 \tag{4.6}$$

Where $\boldsymbol{W}_1 \in \mathbb{R}^{d \times d}$, $\boldsymbol{b}_1 \in \mathbb{R}^d$, $\boldsymbol{W}_2 \in \mathbb{R}^{2 \times d}$, and $\boldsymbol{b}_2 \in \mathbb{R}^2$ are learnable parameters. The model is trained by optimizing the regular cross-entropy classification loss from Equation (2.13).

Hyperparameters In this task, the input sequences are truncated to a maximum length of 800 tokens. During the sweep the batch size h_b is drawn from $\{5, 8, 10, 12, 16\}$, the learning rate h_{lr} is drawn from the interval $[10^{-5}, 10^{-4}]$, and the maxi-

```c
static void vc1_inv_trans_8x8_dc_c(uint8_t *dest, int
    linesize, DCTELEM *block)
{
    int i;
    int dc = block[0];
    const uint8_t *cm;
    dc = (3 * dc +  1) >> 1;
    dc = (3 * dc + 16) >> 5;
    cm = ff_cropTbl + MAX_NEG_CROP + dc;
    for(i = 0; i < 8; i++){
        dest[0] = cm[dest[0]];
        dest[1] = cm[dest[1]];
        dest[2] = cm[dest[2]];
        dest[3] = cm[dest[3]];
        dest[4] = cm[dest[4]];
        dest[5] = cm[dest[5]];
        dest[6] = cm[dest[6]];
        dest[7] = cm[dest[7]];
        dest += linesize;
    }
}
```

Figure 4.9: Example of a code snippet with a vulnerability from the devign dataset (Zhou et al. 2019).

mum number of epochs h_e is drawn from $\{3, 5, 8, 10\}$. The sweep optimizes validation accuracy.

Clone Detection

The Poj-104 dataset has been introduced by Mou et al. (2016). It consists of C and C++ programs for 104 problems sourced from student submissions to a pedagogical programming open judge system (Lu et al. 2021). For each problem 500 solutions are part of the dataset, so that in total the dataset contains 52,000 samples. CodeXglue defines the clone detection task for this dataset as a retrieval task, where the model is tasked to retrieve other code snippets that belong to the same problem as the given code snippet. Correspondingly, the task is evaluated using a retrieval metric: the MAP@499 metric, which is the mean average precision obtained by analyzing the first 499 retrieved samples. Definitions can be found in Equations (2.29) and (2.30). CodeXglue defines a train, validation and test split for the dataset based on the problem level, which is important since it ensures proper generalization to unseen problems. The training set consists of 64 problems, the validation set of 16 problems, and the test set of 24 problems. Examples of the dataset are shown in Figure 4.10.

```c
int main()
{
  char str[105];
  memset (str, 0, sizeof(str));
  cin.getline(str, 105);
  str[strlen(str)] = str[0];
  for (int i = 0; i < strlen(str) - 1; i++)
    cout <<(char)(str[i] + str[i+1]);
  return 0;
}
```

(a) Code snippet from problem 91.

```c
int main()
{
  char a[105], * p, x;
  int i = 0;
  gets(a);
  p = a;
  x = * p;
  for(i = 0; * (p + i + 1) != '\0'; i++)
    * (p + i) += * (p + i + 1);
  * (p + i) += x;
  cout << a;

  return 0;
}
```

(b) Code snippet from problem 91.

```c
int main()
{
  int count(int facevalue,int sum,int n);
  int n,sum=0;
  cin>>n;
  int facevalue[6]={100,50,20,10,5,1},num[6];
  for(int i=0;i<6;i++)
  {
    num[i]=count(facevalue[i],sum,n);
    cout<<num[i]<<endl;
    sum+=num[i]*facevalue[i];
  }
  return 0;
}

int count(int facevalue,int sum,int n)
{
  for(int i=0;;)
  {
    if((i+1)*facevalue+sum<=n)
      i++;
    else
      return i;
  }
}
```

(c) Code snippet from problem 97.

```c
int main()
{
  int i,er=0,ws=0,sh=0,wu=0,b=0,money;
  cin>>money;
  for(;;)
  {if(money/100>=1) {money=money-100;b=b+1;}
  else break;}
  for(;;)
  {if(money/50>=1) {money=money-50;ws=ws+1;}
  else break;}
  for(;;)
  {if(money/20>=1) {money=money-20;er=er+1;}
  else break;}
  for(;;)
  {if(money/10>=1) {money=money-10;sh=sh+1;}
  else break;}
  for(;;)
  {if(money/5>=1) {money=money-5;wu=wu+1;}
  else break;}
  cout<<b<<endl<<ws<<endl<<er<<endl<<
  ↪  sh<<endl<<wu<<endl<<money<<endl;
  return 0;
}
```

(d) Code snippet from problem 97.

Figure 4.10: Examples of code snippets from the clone detection dataset (Mou et al. 2016). The task is to retrieve code snippets that belong to the same problem as the given code snippet. For example, given the code snippet in (a), the model should retrieve the code snippet in (b) and not the code snippets in (c) and (d).

Model Since the clone detection task is a retrieval task, the model architecture needs to be adapted for contrastive learning. As for the defect detection task, the encoder-part of the model is used to encode the code snippets into a fixed-size representation, by using the hidden state of the language identifier token of the encoder's last layer. The model is trained using the InfoNCE loss from Equation (2.19) to maximize the similarity between the query code snippet and other sequences from the same problem, while minimizing the similarity to code snippets from other problems. The similarity between two code snippets is computed using the cosine similarity between their embeddings. Please refer to Section 2.2.2 for more details on contrastive learning.

Hyperparameters With the sweep the validation MAP@499 is optimized. The input sequences are truncated to a maximum length of 800 tokens. The batch size h_b—which has an influence to the number of negative samples during contrastive training, since in-batch negatives are used—is drawn from $\{8, 12, 16, 20\}$, the learning rate h_{lr} is drawn from the interval $[10^{-5}, 10^{-4}]$, and the maximum number of epochs h_e is drawn from $\{5, 8, 10\}$. The non-pretrained model has the additional option to train for 15 epochs.

4.6 Results

This section presents the results of the experiments conducted on CodeXGlue.

4.6.1 Comparison of Structural and Regular Pretraining

> *Does structural pretraining provide a stronger learning signal than regular pretraining?* – RQ 4.1

The results in Table 4.3 show that the models trained with structural pretraining either outperform or match the performance of regular pretrained models consistently across all evaluated tasks. In particular, it can be seen that

- in the code translation task, SyntaxPT outperforms the regular model by 2.9 p.p. in EM,
- in the code refinement task, structural pretraining results in a 0.7 p.p. improvement in EM on the small subset and a 1.8 p.p. improvement for the medium subset,
- for code summarization, SyntaxPT performs marginally better than the regular model (+0.1 BLEU),
- for defect detection, SyntaxPT matches the accuracy of the regular model,
- and for clone detection, SyntaxPT outperforms the regular model by 5.7 p.p. in MAP@499.

Pretraining	Translation (EM)		Refinement (EM)		Summarization (BLEU)	Defect Detection (Accuracy)	Clone Detection (MAP)
	Java→C#	C#→Java	Medium	Small			
RegularPT	63.7	66.2	14.9	22.1	19.3	**68.2**	83.8
SyntaxPT	**66.8**	**68.9**	**16.7**	**22.8**	**19.4**	**68.2**	**89.5**

Table 4.3: Comparison of the regular and structural pretrained models on the CodeXGlue tasks. All metrics are provided in percentages and the best result for each task is highlighted in bold.

On three out of five tasks SyntaxPT outperforms the regular model by a significant margin. The consistent improvements across all tasks indicate that the structural pretraining indeed provides a stronger learning signal than regular pretraining. However, on two tasks SyntaxPT performs equally well or marginally outperforms the regular model. For code summarization the narrow margin can be attributed to the large size of the dataset, which seems to provide a sufficient learning signal for both models itself. Similarly, the defect detection task does not seem to benefit from structural pretraining. An explanation could be that reasoning about identifier names may not be a strong indicator for detecting defects.

Noteworthy is the large improvement in the clone detection task, where SyntaxPT outperforms the regular model by 5.7 p.p. in MAP@499. The improvement feels natural since SyntaxPT—which needed to learn a representation for code without the ability to rely on identifier names—is good at identifying code clones, which are often similar in structure but differ in identifier names. Also, another thing about the clone detection task is particularly interesting: during validation the regular model performed better than SyntaxPT, but at test time it falls far behind. The regular model dropped from 89.6% validation MAP to 83.8% test MAP, while SyntaxPT achieved 88.4% MAP during validation and 89.5% MAP during testing. This large drop in performance indicates that the regular model overfits to the validation set. We analyzed the variance of the three best runs of each hyperparameter sweep, and a similar pattern emerged: the regular model had an average drop of 8.8 p.p. MAP (3.1 p.p. stddev), while SyntaxPT had an average drop of only 0.7 p.p. (stddev of 2.1 p.p.). This indicates that SyntaxPT is better at generalizing to unseen data, at least in the clone detection task.

The better performance of SyntaxPT indicates that structural pretraining indeed creates more challenging training samples from the same data. Another indicator about the strength of the learning signal is the pretraining loss. Loss values near zero indicate that the model can predict the masked tokens with high confidence, while higher loss values indicate that the model struggles to predict the masked tokens. Not surprisingly, the training loss of the structural pretrained models is on average 0.2 higher than the regular pretrained models, as shown in Figure 4.11. This is due to the fact that regular masking often targets simpler elements, such as whitespace or keywords.

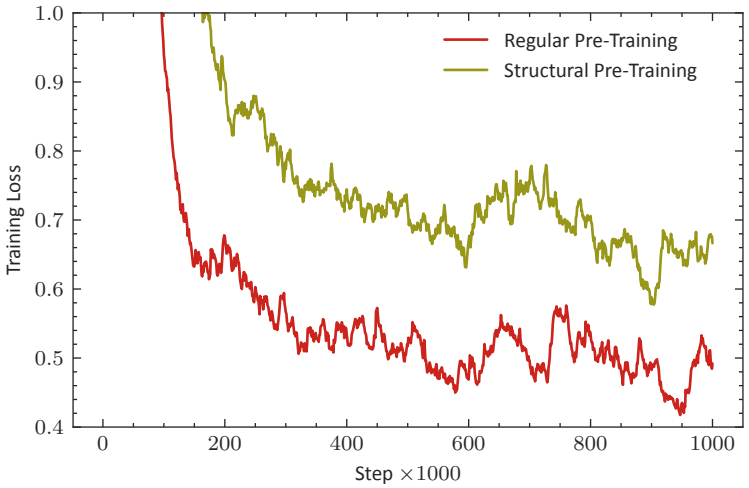


Figure 4.11: Comparison of the training losses of the structural (green) and regular pretrained models (red). The structural pretraining leads to higher training losses, which indicates that the model struggles more to predict the masked tokens.

Despite the lower difficulty of regular pretraining, models trained with structural pretraining achieve a higher test performance on fine-tuning tasks. Furthermore, not only the test performance is higher, but also the training is more stable, as shown before on the clone detection task. An alternative perspective on the stability of the training is visualized in Figure 4.12 by plotting the validation performance of the models in the hyperparameter sweeps. The average performance of the structural pretrained models is shown in green, while the regular pretrained models are shown in red and the light area represents the 95% confidence interval. It can be seen that the structural pretraining on average has a higher performance than the regular pretraining on all tasks. Additionally, the confidence interval for the structural runs is much smaller than the one for the regular runs, which indicates that structural pretraining leads to more stable fine-tuning performance and less sensitivity with respect to hyperparameters.

These findings support the argument that structural tasks present more challenging problems, which require deeper reasoning about the code and thereby provides a stronger learning signal. The model trained with structural pretraining is better at generalizing to unseen data and is more stable during fine-tuning to downstream tasks.

4.6.2 *Benefit of Pretraining on Code*

> *What is the benefit of pretraining on code compared to training from scratch?* – RQ 4.2

Following recent lines of research that demonstrate the benefits of pretraining on large datasets (Devlin et al. 2019; Feng et al. 2020; Guo et al. 2021) the structural pretrained

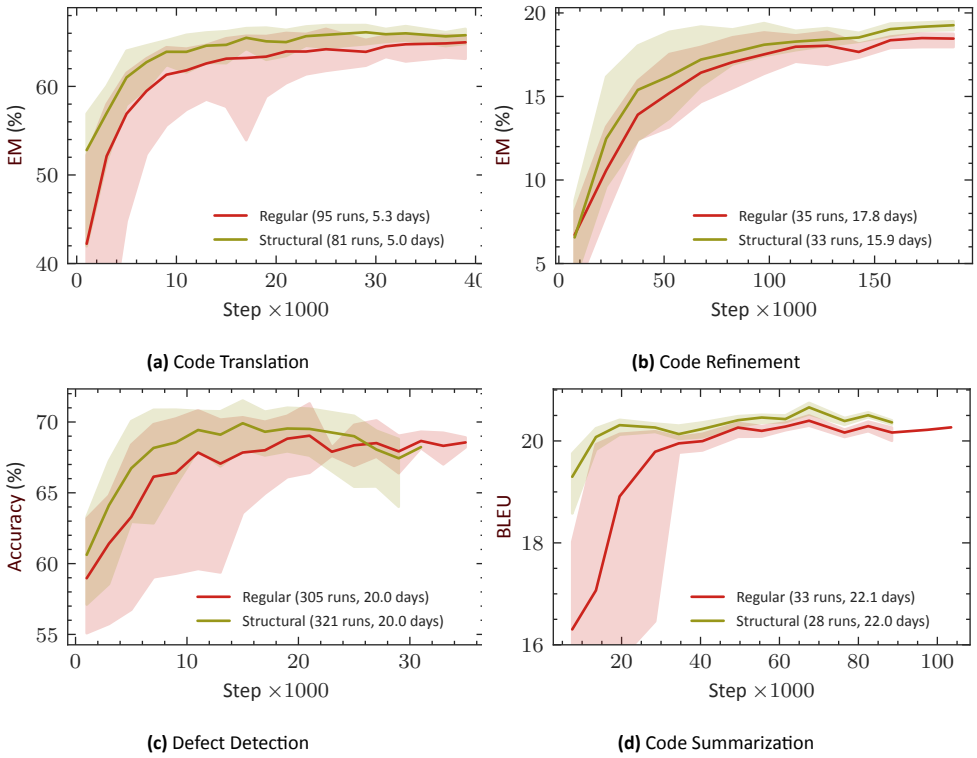


Figure 4.12: Comparison of the validation performance on the fine-tuning tasks for the structural and regular pretrained models at each training step. The results of every run in the sweep are aggregated and the average performance is shown as a solid line, while the light area represents the 95% confidence interval.

model is compared to a model which is trained end-to-end on the CodeXGlue tasks in Table 4.4. The clone detection task benefits strongly from pretraining, with a 69% relative improvement in MAP@499. The subjective variability in identifier names makes keyword matching difficult, and this variability seems to be difficult to learn end-to-end from the training data.

Additionally, it can be observed that the two generation tasks code translation and code refinement benefit significantly from pretraining, with an average relative improvement of 146% in EM on the translation task and 86% on the refinement task. This is in line with previous findings that pretraining can significantly improve the performance of models on downstream tasks (Liu et al. 2019; Wang et al. 2021b).

Especially, the decoder part of the model benefits from pretraining, as the training datasets may simply not be large enough to learn the complex patterns of code and natural language needed to generate the correct output. But even for tasks with a large training set—such as the code summarization task, that contains over 900k training samples—an absolute im-

Pretraining	Translation (EM)		Refinement (EM)		Summarization (BLEU)	Defect Detection (Accuracy)	Clone Detection (MAP)
	Java⟶C#	C#⟶Java	Medium	Small			
Transformer (no PT)	28.0	27.2	7.4	15.7	17.7	63.7	52.9
SyntaxPT	**66.8**	**68.9**	**16.7**	**22.8**	**19.4**	**68.2**	**89.5**

Table 4.4: Comparison of the structural pretrained model to a model that is trained end-to-end on the CodeXGlue tasks. All metrics are provided in percentages and the best score for each task is highlighted in bold.

Model	Ruby	JS	Go	Python	Java	PHP	All
Rst	14.8	15.0	18.6	17.9	18.6	23.8	18.1
Transformer (no PT)	13.9	14.6	18.1	18.2	18.2	23.1	17.7
RegularPT	**15.9**	15.8	19.3	19.2	**20.0**	25.5	19.3
SyntaxPT	15.8	**15.9**	**19.3**	**19.4**	19.7	**26.0**	**19.4**

Table 4.5: Comparison of the Rst model from Chapter 3 to the pretrained models on the CodeSearchNet dataset.

provement of 1.7 p.p. BLEU can be observed. Overall, one can conclude that pretraining on code provides a significant benefit for all code understanding benchmarks.

4.6.3 Structural Pretraining vs. Relative Structural Transformer

How does the structural pretraining compare to the Rst model from Chapter 3? – RQ 4.3

The Rst model from Chapter 3 is compared to the pretrained models on the CodeSearchNet dataset. The results are shown in Table 4.5. The pretrained models outperform the Rst model on all languages, but it is important to note, that the Rst model was trained end-to-end on the CodeSearchNet dataset. While the internal structural bias of the Rst model is beneficial compared to no pretraining, any type of pretraining provides better performance. During pretraining the model sees a variety of code snippets with their documentations, and the model can internalize this information to generate better code summaries.

4.6.4 Comparison with State-of-the-Art

How do the models perform compared to other state-of-the-art models? – RQ 4.4

Next, the models are compared to other state-of-the-art models on the CodeXGlue benchmark (Lu et al. 2021). The scores of related work are taken from the respective paper or the CodeXGlue leaderboard, unless otherwise stated[7]. Note that these experiments have been performed in 2021, and comparisons in this section are based on state-of-the-art models at that time, with a cutoff date of February 2022. Please refer to the CodeXGLUE

[7] The leaderboard can be accessed at https://microsoft.github.io/CodeXGLUE/ (last accessed 09/2024).

Model	Java→C#		C#→Java	
	BLEU	EM	BLEU	EM
Naive Copy	18.5	0	18.7	0
RoBERTa-code	77.5	56.1	72.0	57.9
CodeBert	79.9	59.0	72.1	58.8
GraphCodeBert	80.6	59.4	72.6	58.8
SynCobert	80.8	60.4	76.5	61.3
PLBart	83.0	64.6	78.4	65.0
CodeT5	84.0	65.9	79.9	66.9
2024 Leaderboard[a]	85.0	66.6	80.7	67.7
Transformer (no PT)	48.7	28.0	41.0	27.2
RegularPT	83.5	63.7	79.8	66.2
SyntaxPT	**85.4**	**66.8**	**81.2**	**68.9**

Table 4.6: Comparison against the state-of-the-art on the code translation task.

leaderboard for the most recent results and models. As a reference, the best result from the leaderboard at the time of the writing of the thesis is added to each table, without explicitly mentioning it in the text. Many approaches propose different variants of their model. For example, the CodeT5 model is evaluated with and without a multi-task fine-tuning approach. In that case, the best performing variant is reported in the tables, which does not necessarily have to be the same variant for all tasks. Please refer to the original paper to obtain the exact variant.

Code Translation

When comparing the models to the state-of-the-art on the code translation task in Table 4.6, it can be seen that the structural pretrained model outperforms all other models on both translation directions. Compared to the state-of-the -art CodeT5 model SyntaxPT achieves a +1.4 p.p. BLEU and a +0.9 p.p. EM improvement when translating from Java to C#, and +1.3 p.p. BLEU and +2 p.p. EM improvement when translating from C# to Java. Interestingly, the regular model performs similar, but slightly worse than the CodeT5 model, which is surprising, as the CodeT5 model has been trained on a mixture of regular and structural tasks. Even though the models trained in this chapter have been trained on more data, the regular trained model could not outperform the same-sized CodeT5 model. This is another indicator that the structural pretraining may provide a stronger learning signal than regular pretraining.

Now some qualitative examples of SyntaxPT are shown. The model successfully translated the code shown in Figures 4.6 and 4.13. Thereby, the model correctly distinguishes

[a] Results belong to model proposed by Tipirneni et al. (2024).

```java
public RuleTagToken(String ruleName, int bypassTokenType, String label){
  if (ruleName == null || ruleName.isEmpty()){
    throw new IllegalArgumentException("ruleName cannot be null or empty.");
  }
  this.ruleName = ruleName;
  this.bypassTokenType = bypassTokenType;
  this.label = label;
}
```

(a) Java code snippet.

```cs
public RuleTagToken(string ruleName, int bypassTokenType, string label){
  if (string.IsNullOrEmpty(ruleName)){
    throw new ArgumentException("ruleName cannot be null or empty.");
  }
  this.ruleName = ruleName;
  this.bypassTokenType = bypassTokenType;
  this.label = label;
}
```

(b) C# code snippet.

Figure 4.13: This function was translated correctly in both directions.

```cs
public static SpatialStrategy GetSpatialStrategy(int roundNumber){
  SpatialStrategy result;
  if (!spatialStrategyCache.TryGetValue(roundNumber, out result) || result == null){
    throw new InvalidOperationException("Strategy should have been init'ed by SpatialDocMaker by now");
  }
  return result;
}
```

(a) C# code snippet.

```java
public static SpatialStrategy getSpatialStrategy(int roundNumber){
  SpatialStrategy result = spatialStrategyCache.get(roundNumber);
  if (result == null) {
    throw new IllegalStateException("Strategy should have been init'ed by SpatialDocMaker by now");
  }
  return result;
}
```

(b) Correct predicted Java code snippet.

```cs
public static SpatialStrategy GetSpatialStrategy(int roundNumber){
  SpatialStrategy result = spatialStrategyCache.Get(roundNumber);
  if (result == null){
    throw new InvalidOperationException("Strategy should have been init'ed by SpatialDocMaker by now");
  }
  return result;
}
```

(c) Incorrect predicted C# code snippet.

Figure 4.14: While this function was translated correctly from C# to Java (a)→(b), the translation from Java to C# (b)→(c) did not use the C# specific `TryGetValue` method (see ground truth in (a)).

Model	Refinement	
	Small	Med
Naive Copy	0	0
T5	15.3	4.1
RoBERTa-code	15.9	4.1
CodeBert	16.4	5.2
GraphCodeBert	17.3	9.1
PLBart	19.2	9.0
CodeT5	22.6	14.2
CoTexT	22.6	15.4
2024 Leaderboard	24.0[b]	15.4[c]
Transformer (no PT)	15.7	7.4
RegularPT	22.1	14.9
SyntaxPT	**22.8**	**16.7**

Table 4.7: Exact Match for the code refinement task on the small and medium dataset.

between language specific idioms, such as the `IsNullOrEmpty` check in C# in Figure 4.13 line 2, or rather uses the `foreach` loop in C# line 3 of Figure 4.6. In Figure 4.14, when translating from C# to Java, the model correctly replaces the C#-specific `TryGetValue` method with the Java-specific `get` method, which returns `null` if the key is not present in the map. However, when translating from Java to C# it tries to adapt the pattern from Java to C#, which is incorrect, as a `Get` method does not exist in C#. Either using the `TryGetValue` method or an itemized access with a `ContainsKey` check would be semantically correct translations. Furthermore, the model sometimes adds unnecessary whitespace or brackets in single line if-statements, which are not present in the reference translation. For EM, these changes are considered errors, even though they do not change the semantics of the code. Interestingly, when evaluating the predictions without considering whitespace or brackets, the model achieves an accuracy of 74.5% (+5.6 p.p.) for C# to Java and 68.9% (+2.1 p.p.) for Java to C#.

Code Refinement

The results for the code refinement task are shown in Table 4.7. The structural pretrained model outperforms all other models on both the small and medium dataset. The strongest improvement can be observed on the medium dataset, where the model achieves a 2.5 p.p. improvement in accuracy compared to the CodeT5 model. While there is still an improvement on the small dataset, the difference to CodeT5 is less strong with only 0.2 p.p.. Interestingly, the non-pretrained baseline performs similar or even better on the medium dataset than the pretrained RoBERTa-code and CodeBert models. An explanation could be that those encoder-only models still need to train the decoder

[b] Result belongs to model proposed by Hu et al. (2022).
[c] Result belongs to model proposed by Phan et al. (2021).

Model	Ruby	JS	Go	Python	Java	PHP	All
RoBERTa-code	11.2	11.9	17.7	18.1	16.5	24.0	16.6
CodeBert	12.2	14.9	18.1	19.1	17.7	25.2	17.8
Rst (Chapter 3)	14.8	15.0	18.6	17.9	18.6	23.8	18.1
t5	14.2	14.6	19.2	19.3	18.4	24.6	18.4
CoTexT	14.0	15.0	18.9	19.7	19.1	24.6	18.6
PLBart	14.1	15.6	18.9	19.3	18.5	23.6	18.3
CodeT5	15.7	**16.2**	**19.8**	**20.4**	**20.5**	**26.1**	**19.8**
2024 Leaderboard[d]	17.2	18.2	21.6	23.1	22.6	28.8	21.9
Transformer (no PT)	13.9	14.6	18.1	18.2	18.2	23.1	17.7
RegularPT	**15.9**	15.8	19.3	19.2	20.0	25.5	19.3
SyntaxPT	15.8	15.9	19.3	19.4	19.7	26.0	19.4

Table 4.8: Code summarization results on the CodeSearchNet dataset. As Feng et al. (2020) this table reports smoothed cumulative BLEU-4 scores.

from scratch during fine-tuning. This gave the models trained in this chapter a head start, because the pretrained decoder needed to be fine-tuned to the specific use-case.

Examples of correctly and incorrectly predicted fixes are shown in Figures 4.15 and 4.16. The model correctly learns that unused variables, and unnecessary print or if statements can be removed or replaced by simpler terms. It does not simply learn that unused variables need to be removed, in the fourth example the model detects that an unused parameter needs to be used inside a method and correctly applies that fix. It can also detect that it should be first checked if an array has been initialized before item access is performed. Additionally, it detects uncommon loop conditions (e.g., `i != VAR_1`) and uses the more common and easier to understand syntax. On the other hand only about a fifth of the examples were predicted correctly and Figure 4.16 shows three incorrectly predicted fixes. In the first example the model correctly removes the off-by-one error, however, it also retains the original behavior of returning `null` when `i == 0`. This special case logic can be intended and there is no way of figuring this out for the model, given the limited information available. The same counts for the second example, where the model incorrectly changes the access level modifier from `protected` to `public` instead of `private`. In some examples, such as the third example, the fix may not be reasonably predicted. The model removes the unnecessary print statement, however, the reference translation asks for saving the parameter `VAR_1` in a variable `p`.

Code Summarization

On code summarization the model was evaluated on the CodeSearchNet dataset. The detailed results are shown in Table 4.8. As already discussed in RQ 4.1 both pretraining strategies perform similar, with SyntaxPT slightly outperforming the regular model.

[d] Results belong to model proposed by Li et al. (2023).

```java
boolean METHOD_1 ( java.lang.String text ) { java.util.regex.Matcher VAR_1 = VAR_2 . METHOD_2 ( text ) ; if ( VAR_1 . METHOD_3 ( ) ) { return true ; } else { return false ; } }
boolean METHOD_1 ( java.lang.String text ) { java.util.regex.Matcher VAR_1 = VAR_2 . METHOD_2 ( text ) ; return VAR_1 . METHOD_3 ( ) ; }
```

(a) Remove redundant if statement.

```java
public java.util.List < TYPE_1 > METHOD_1 ( int VAR_1 ) { return VAR_2 . METHOD_1 ( ) ; }
public java.util.List < TYPE_1 > METHOD_1 ( int VAR_1 ) { return VAR_2 . METHOD_1 ( VAR_1 ) ; }
```

(b) Add missing argument.

```java
private void METHOD_1 ( ) { if ( ( VAR_1 . length ) > 1 ) { VAR_2 = ( VAR_1 . length ) - 1 ; METHOD_2 ( VAR_1 [ VAR_2 ] ) ; } }
private void METHOD_1 ( ) { if ( ( VAR_1 ) != null ) { if ( ( VAR_1 . length ) > 1 ) { VAR_2 = ( VAR_1 . length ) - 1 ; METHOD_2 ( VAR_1 [ VAR_2 ] ) ; } } }
```

(c) Add check for null.

```java
private void METHOD_1 ( int VAR_1 , char c ) { for ( int i = 0 ; i != VAR_1 ; i ++ ) { out . METHOD_2 ( c ) ; } out . METHOD_3 ( ) ; }
private void METHOD_1 ( int VAR_1 , char c ) { for ( int i = 0 ; i < VAR_1 ; i ++ ) { out . METHOD_2 ( c ) ; } out . METHOD_3 ( ) ; }
```

(d) Change comparison operator.

Figure 4.15: Correctly predicted fixes. The first line is the buggy code snippet, and the second line is the correct fix.

```java
public TYPE_1 METHOD_1 ( int i ) { if ( i == 0 ) return null ; return VAR_1 . get ( ( i - 1 ) );}
public TYPE_1 METHOD_1 ( int i ) { return VAR_1 . get ( i ) ; }
public TYPE_1 METHOD_1 ( int i ) { if ( i == 0 ) return null ; return VAR_1 . get ( i ) ; }
```

(a)

```java
protected boolean METHOD_1 ( java.lang.String name ) { return VAR_1 . METHOD_2 ( name ) ; }
private boolean METHOD_1 ( java.lang.String name ) { return VAR_1 . METHOD_2 ( name ) ; }
public boolean METHOD_1 ( java.lang.String name ) { return VAR_1 . METHOD_2 ( name ) ; }
```

(b)

```java
public void METHOD_1 ( TYPE_1 VAR_1 ) { java.lang.System.out.println ( ( STRING_1 + ( VAR_1 . getName ( ) ) ) ) ; VAR_2 . METHOD_1 ( VAR_1 ) ; }
public void METHOD_1 ( TYPE_1 VAR_1 ) { java.lang.System.out.println ( ( STRING_1 + ( VAR_1 . getName ( ) ) ) ) ; p = VAR_1 ; VAR_2 . METHOD_1 ( VAR_1 ) ; }
public void METHOD_1 ( TYPE_1 VAR_1 ) { VAR_2 . METHOD_1 ( VAR_1 ) ; }
```

(c)

Figure 4.16: Three incorrectly predicted bug fixes. The first line in each sub-figure is the buggy code snippet, the second line the correct fix, and the third line is the predicted fix.

Model	Accuracy
RoBERTa-code	61.1
Transformer (no PT)	61.6
t5	61.9
CodeBert	62.1
Code2vec	62.5
GraphCodeBert	63.2
PLBart	63.2
SynCobert	64.5
CodeT5	65.8
CoTexT	66.6
2024 Leaderboard[e]	69.3
Transformer (no PT)	63.7
RegularPT	**68.2**
SyntaxPT	**68.2**

Table 4.9: Accuracy of the models on the defect detection task.

Model	MAP
Aroma	55.1
RoBERTa-code	76.7
CodeBert	82.7
GraphCodeBert	85.2
PLBart	86.3
SynCobert	88.2
CodeT5	88.7
2024 Leaderboard[g]	90.5
Transformer (no PT)	52.9
RegularPT	83.8
SyntaxPT	**89.5**

Table 4.10: Results of the code clone detection task. For task descriptions please refer to Section 4.5.7.

In contrast to RQ 4.1, which considered only the overall BLEU score, Table 4.8 also shows detailed results for each language. SyntaxPT outperforms the regular model on JavaScript, Python, and PHP, while the regular model outperforms SyntaxPT on Ruby and Java.

When comparing the models to the state-of-the-art it can be seen, that on Ruby both models achieve a new state-of-the-art score. Furthermore, SyntaxPT achieves the second-best overall BLEU score, but is outperformed by the CodeT5 model, which achieves +0.4 p.p. BLEU. This can be explained by the fact that the CodeT5 model has been trained on additional bimodal data[8], which includes the code summarization task. Thereby, the model has been explicitly trained to translate from both code to natural language descriptions and vice versa, which may strengthen the connection between code and natural language. However, one can conclude that the both pretraining strategies are competitive with the state-of-the-art on the code summarization task, even though they have not explicitly been pretrained on bimodal data.

[e] Results belong to model proposed by Guo et al. (2022).
[f] Results belong to model proposed by Liu et al. (2023).

[8] The CodeT5 model has been pretrained on 3.1M bimodal datapoints. The models trained in this chapter have been fine-tuned only on the 900k bimodal examples from the CodeSearchNet dataset, as shown in Table 4.2.

Defect Detection

The comparison to the state-of-the-art on the defect detection tasks is shown in Table 4.9. The regular and structural pretrained models both achieve an new state-of-the-art accuracy of 68.2%. Thereby, all other models in this comparison are outperformed, including the CodeT5 model and the previous state-of-the-art model CoTexT. This work achieves a significant +2.4 p.p. improvement over the CodeT5 model and a +1.6 p.p. improvement over the CoTexT model.

Also in comparison to other transformer-based LMs like GraphCodeBert, PlBart, and SynCobert which achieve accuracies of 63.2%, 63.2% and 64.5% respectively, the models developed in this chapter demonstrate a substantial improvement. Interestingly, our non-pretrained baseline—that achieves a similar accuracy of 63.7%—is better than previously reported transformer baselines (61.6%) and even outperforms many of the aforementioned code-LMs. This indicates that the preprocessing and tokenization strategy proposed in this chapter may be particularly well-suited for the defect detection task.

Clone Detection

Table 4.10 presents the performance of the models on the clone detection task on the Poj-104 dataset. The structural pretrained model outperforms all other models in the comparison and achieves a new state-of-the-art MAP of 89.5%. Notably, SyntaxPT outperforms the CodeT5 model by 0.8 p.p.. This indicates that the structural pretraining strategy produces not only a strong decoder, but also the encoder retains valuable information for the clone detection task. It produces a representation that provides a better semantic understanding of code and the relationships between code snippets than dedicated encoder-only models like GraphCodeBert or CodeBert. Visualizing qualitative results for the clone detection task is challenging, as most of the results are correct and false positives are mainly on the lower end of the ranking. Instead of showing examples, a confusion matrix is shown in Figure 4.17. The matrix shows the amount of problems in the first 499 search results for each query problem. Ideally the diagonal should be dark, indicating that all 499 other solutions of the same problem have been ranked higher than the solutions of other problems. It can be seen, that some problems tend to get confused more often with other problems, e.g., problem 83 is often confused with problem 99 (72,279 times) and problem 97 is often confused with problem 91 (49,007 times). However, this visualization does not respect the order of the results, a false positive at the first position is counted to the same field as a false positive at the 499th position. To visualize also the order of the results, Figure 4.18 shows a heatmap that visualizes the relative amount of correct samples at each position in the search results for each query problem. It can be seen that some problems, such as 81, 85, and 95, are often correctly retrieved, while others, such as 83, 97, and 104, are more often confused with other problems.

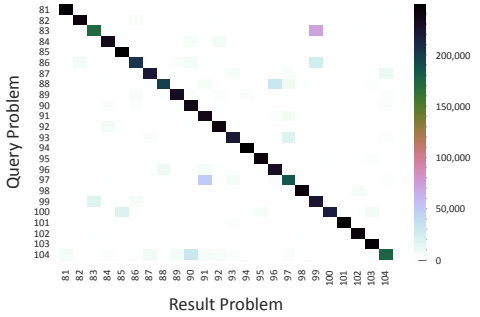


Figure 4.17: This confusion matrix visualizes the amount of problems in the first 499 search results for every problem in the POJ-104 test set. Note that this visualization ignores the order of the results.

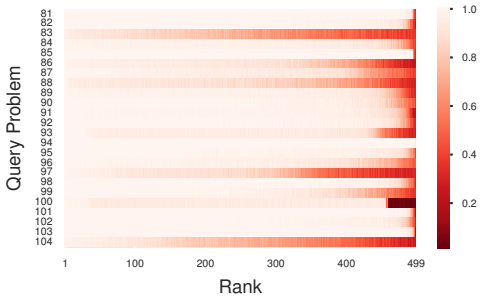


Figure 4.18: Grouped by the query problem, the heatmap shows the relative amount of correct samples at a specific position in the result list. White (1.0) indicates that every result at this position has been correct, while dark red (0.0) indicates that no result at this rank has been correct.

When looking at the examples for this task in Figures 4.10 and 4.19 it can be seen, that similar programs often share similar inputs and outputs. This is expected as the output format may be part of the task given to the students. Two of such shared sub-structures are highlighted in Figure 4.19. Even though the programs are quite different in identifier naming and structure, inputs follow a similar pattern: `scanf("%s",VAR)`, `printf("No")`, and `printf("%c=%d\n",VAR,VAR)`. Intuitively, the subtree-specific training strategy should be well-suited to detect such patterns, and the results in Table 4.10 confirm this intuition.

4.7 Conclusion and Future Work

This chapter investigated the effectiveness of structural pretraining tasks for encoder-decoder transformer LMs in the domain of source code understanding. Building upon the limitations of regular pretraining tasks like MLM and short span masking, a novel structural pretraining task called tree-based span selection was introduced. This task uses the AST to select and mask syntactic segments, which produces challenging and contextually rich training examples. This chapter also extended and improved the structural identifier deobfuscation task proposed by Lachaux et al. (2021), by extending it to also hide method calls and introducing probabilistic masking rates. By combining these structural tasks with regular pretraining tasks in a dynamic multi-task pretraining pipeline, the SyntaxPT model was trained solely on unimodal code data in a self-supervised fashion.

The experimental results demonstrate that structural pretraining provides a stronger learning signal than regular pretraining, which results in improved code understanding capabilities. Specifically, the structural SyntaxPT model consistently outperformed the baseline model trained with regular pretraining tasks across multiple code understanding benchmarks from the CodeXGlue suite (Lu et al. 2021). The structural model also

```c
int main(){
  int i,n,m=0,t=0;
  char k,s[300];
  scanf("%s",s);
  n=strlen(s);
  for(k='a';k<='z';k++){
    for(i=0;i<n;i++){
      if(s[i]==k){
      m++;
      }
    }
```

```c
    if(m!=0){
      printf("%c=%d\n",k,m);
      t++;
    }
    m=0;
  }
  if(t==0){
    printf("No");
  }
  return 0;
}
```

(a) Code snippet for problem 100.

```c
int main () {
  char as[300];
  scanf("%s",as);
  int sz[300]={0};
  int len=strlen(as);
  char sd[300],df[300];

  int e=0;
  for(int i=0;i<len;i++){
    int w=as[i];
    if((w<=122)&&(w>=97)){
      sd[e]=as[i];
      e++;
    }
  }
  for(int s=1;s<=e;s++){
    for(int q=0;q<e-s;q++){
      int m,n;
      char y;
      m=sd[q];n=sd[q+1];
```

```c
      if(m>n){
        y=sd[q];
        sd[q]=sd[q+1];
        sd[q+1]=y;
      }
    }
  }
  if(e!=0){
    df[0]=sd[0];sz[0]=1;
  int t=0;
  for(int k=1;k<e;k++){
    char r=sd[k];
    int a=0;
    int d;
    for(int j=0;j<=t;j++){
      if(r==df[j]){

        a++;
        d=j;
      }
```

```c
      }
    }
      if(a!=0){
        sz[d]++;
      }else {
        t++;
        df[t]=r;
        sz[t]++;
      }
    }

  for(int h=0;h<=t;h++){
    printf("%c=%d\n",df[h],sz[h]);
  }
  }
  if(e==0){
    printf("No");
  }
  return 0;
}
```

(b) Another code snippet for problem 100.

Figure 4.19: Similar problems can often be identified by matching sub-structures in the code snippets, as indicated by the highlighted lines in (a) and (b), which is similar to *tree-based span selection*.

showed more stable and consistent fine-tuning performance, indicating better generalization to unseen data. Furthermore, this chapter confirmed findings of related work that pretraining on code offers significant benefits compared to training models from scratch (Feng et al. 2020). SyntaxPT not only outperformed the end-to-end trained model but also achieved new state-of-the-art results at the time of the experiments on several tasks, including code translation, code refinement, and defect detection.

While the approach developed in this chapter has demonstrated significant improvements, it is not without limitations. Due to computational constraints, the author was unable to explore the effects of larger model sizes, which could potentially amplify the benefits of structural pretraining. Future work could investigate whether the improvements observed with structural pretraining persist with larger model architectures. Additionally, an ablation study to isolate the individual contributions of each structural task was not feasible within our computational constraints (a pretraining run takes more than 3 weeks). Understanding the specific impact of each task is left for future work. Moreover, the identifier deobfuscation task, as extended in this chapter, has the potential to be applied in practical applications such as variable renaming and estimating identifier quality. This thesis will explore using the model's likelihoods to estimate identifier quality in Chapter 7.

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it.

— Samuel Johnson, 1775

5

Contrastive Pretraining for Contextualized Code Search

The last two chapters focussed on improving the code understanding capabilities of generative transformers by introducing a syntactical prior into the transformer model. This chapter aims to further explore the retrieval capabilities of SyntaxPT, by developing a novel approach for training a code retrieval model in a self-supervised manner.

5.1 Introduction and Motivation

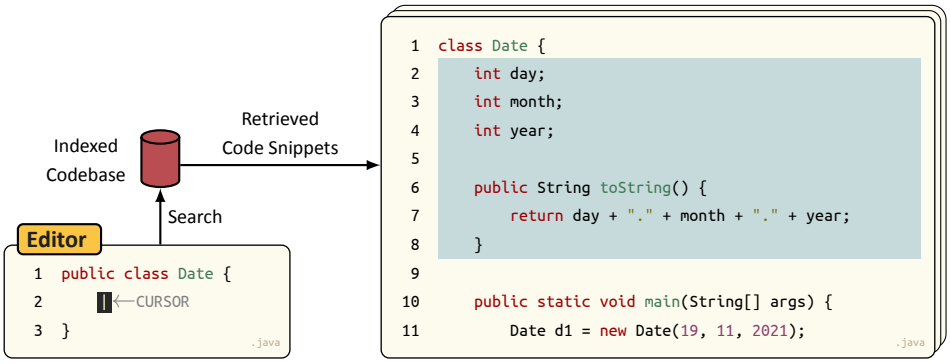
The strong semantic understanding of code by LMs trained on source code—such as the one introduced in the last chapter—has started to influence software development (Lu et al. 2021). Models and tools for code generation, such as GitHub's Copilot and ChatGPT (Svyatkovskiy et al. 2020; Chen et al. 2021; GitHub 2024; OpenAI 2024a), are widely used nowadays in 2024. These tools have been made possible by the fact, that increasing the amount of model parameters leads to even better generational performance (Kaplan et al. 2020), as seen in large models like GPT-3, which has 175 billion parameters (Brown et al. 2020), over 700 times more than the 245 million parameter SyntaxPT model studied in this thesis. However, generative LMs of such size require high compu-

This chapter is adapted from **Johannes Villmow**, Viola Campos, Adrian Ulges, and Ulrich Schwanecke (2022). Addressing Leakage in Self-Supervised Contextualized Code Retrieval. In *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*. International Committee on Computational Linguistics, pp. 1006–1013, licensed under CC BY 4.0.

tational and environmental costs to train and operate. A single training run of such models is estimated to emit more carbon dioxide as 125 round-trip flights between New York and Beijing (Dhar 2020). Even with such large parameter counts, code generation tools often produce subtle bugs or hallucinations (Maynez et al. 2020), which has led some projects, such as NetBSD, to ban code generated by LMs (NetBSD 2024). Furthermore, the code generated by these models may be subject to copyright claims. To address this, GitHub experiments with matching code generated by Copilot against open-source code to obtain licenses in a post-processing step (Salva 2023). Furthermore, code generation models often lack knowledge about project internals, which limits their application in corporate software development environments. Those key problems with generative models come from the fact that the model generates code in a non-transparent process, out of its own parameters. The source of the knowledge, or the reference implementation from which a proposed solution is derived, is never made transparent. This motivates approaches towards discovering said source, i.e., approaches towards *searching* for relevant code.

In addition to code generation, applications such as natural language code search (Husain et al. 2019) and code clone detection (Svajlenko and Roy 2015) also benefit from the code understanding capabilities of pretrained transformer LMs, as shown in the last chapter. This chapter focuses on code search, which unlike code generation, does not suffer from copyright claim or subtle bugs issues. Here, the source and context of the retrieved code snippet is known, which allows the developer to validate its plausibility, and—when the developer trusts the source—safely reuse the code. Most previous work focuses on so-called natural language code search, where the developer formulates a query in natural language and the system retrieves code snippets that semantically fit to the query (mostly on a function-level). However, while natural language code search is a promising tool for more code reuse, it requires manual query formulation by the developer. However, reuse often fails when developers are unaware of the existence of a helpful implementation that could assist them, and thus not even attempts to search (Frakes and Fox 1996).

This suggests a "queryless" search system, similar to code autocompletion, that integrates seamlessly into the developer's workflow. Such a system would suggest relevant code snippets based on the current coding context, i.e., the code and cursor position at which the developer is currently coding. To this end, the system has to derive the developer's intent and informational needs without explicit input. This approach could minimize the effort required for the developer to search, while at the same time maintaining the benefits from code search. This approach—referred to as Contextualized Code Search (CCS)—is the focus of this chapter.



(a) Query. **(b)** Result 1: `109/02_Date.java`

Figure 5.1: Example of the Contextualized Code Search (CCS) task. In (a) a user is editing a Java class `Date` and the cursor is at the position of the pipe symbol (black). This context is the query for the CCS system. Code snippets that fit at this position are retrieved from the codebase. The highlighted lines in (b)—instance attributes and a `toString` method—have been found relevant to the query and are presented with some context code.

5.1.1 The Contextualized Code Search Task

The task is illustrated in Figure 5.1, which aims to *retrieve* code snippets from a codebase given the developer's current coding context and a position of interest, i.e., such as an open file in a code editor, like an IDE, is called Contextualized Code Search (CCS). It has rarely been studied until now, only a few papers exist (Mishne et al. 2012; Mukherjee et al. 2020; Dahal et al. 2022). In CCS an *encoder*, also called *retriever*, mines a trusted source, such as the company's own codebase, for relevant code fragments (or code snippets). We refer to these fragments as *results* or *targets*. To retrieve these snippets, we define the query as the current coding context with a marked position of interest, e.g., the cursor position in an IDE. Snippets that "fit" at the current cursor position are considered to be relevant. In the example the cursor is inside the class, so the system returns instance attributes and a `toString` method that are relevant to the class `Date` in Figure 5.1b. In other, scenarios relevant snippets could range from single, over multiple lines to larger for-loops.

CCS can be integrated neatly into the developer's workflow, just as code autocompletion: While editing, the user simply clicks a search button and is suggested code snippets fitting at his/her current cursor position. When all indexed code snippets come from trusted sources, the discovered code is guaranteed to be correct, and without copyright issues. CCS enables easy reuse on demand for the developer.

Even though CCS is similar to the aforementioned tasks, it has some distinct differences:

- While *natural language code search* only searches with a natural language query and thus lacks the code context in the editor, CCS may—but does not have to—include a comment describing the missing step, i.e., CCS can be considered a more general

task. Additionally, natural language code search often operates at a fixed granularity (e.g, on a function-level), while CCS aims to retrieve snippets of varying sizes.

- CCS is also very different to *code completion*, where a model would try to synthesize the missing piece from its internal parameters. In CCS the selection and integration of a relevant code snippet is up to the developer, which makes the coding process and responsibilities more transparent and reduces the risk of subtle bugs.
- *Code clone detection* aims to find semantically similar pieces of code, such as code with high overlap to the given one. In clone detection full code files or methods are compared against each other. In contrast, CCS targets code that *completes* a partial query, rather than identifying duplicate code segments.

Challenges The goal of this chapter is a novel process for training a CCS model. This poses several interesting challenges from a machine learning perspective:

1. The query is incomplete, which requires the model to infer the intended semantics from an incomplete description. While retrieving snippets that are structurally similar to the current context, as in clone detection, is relatively straightforward, in CCS it is necessary to retrieve snippets that align with the missing components. Note that code generation faces the same challenge, where it is often approached by the combination with natural language, e.g., by adding a comment that specifies the users intent. The same would be possible with CCS, if the user explains his/her information need in a short comment.
2. The results originate from existing code files and thus use different identifier names or format than the current context. Since keyword search fails in such cases, a semantic understanding of code is required.
3. Last and most importantly, relevancy assessments in form of query-result pairs, which would facilitate a supervised training, are scarce and not available in large numbers[1]. The lack of available labels incentivizes a self-supervised approach towards CCS.

5.1.2 Contributions

This chapter presents a self-supervised approach towards CCS training that bootstraps query-result pairs for contrastive training using only unlabeled pieces of code, which obviously are widely available. Inspiration comes from recent approaches in natural language processing (Lee et al. 2019; Conneau et al. 2020) that learn passage retrieval models for question answering in a weakly supervised manner by solving an inverse cloze task. Instead

[1] For reference, the most commonly used dataset for passage retrieval in NLP, is called the MS MARCO dataset. It consists of 1M questions and 9M passages (Nguyen et al. 2016).

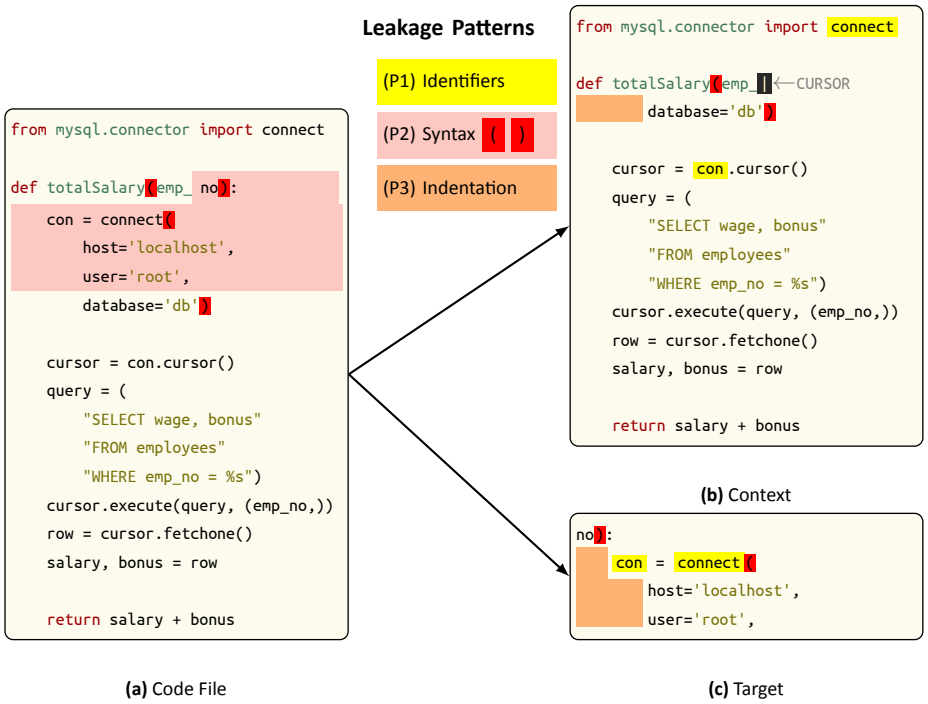


Figure 5.2: Naively bootstrapping context-target pairs for self-supervised CT-based Contextualized Code Search (CCS) is prone to leakage, which are trivial patterns that the encoder can exploit during pair matching, that hinder the learning of semantic similarity. This chapter addresses three types of leakage: (P2) a naive cut splits syntactic constructs (visualized in red), (P1) both context and target share the same identifiers (visualized in yellow), whereas real solutions might have different identifiers, (P3) the same indentation level (visualized in orange).

of predicting a word or sentence given its context, like in the cloze task (Taylor 1953) or MLM, the authors treat a sentence as a pseudo-question and predict which passage it originates from. Following this idea, the approach developed in this chapter learns CCS by erasing random blocks of code instead of sentences from a file. The erased block is a possible result, and the file context forms the query, in which the target block has been replaced by a single token indicating the position of interest. Since in CCS the context is the query and the erased block the result, this is the regular cloze task, and the proposed approach will be referred to as cloze task-based retrieval pretraining. To the best of the author's knowledge, this type of pretraining has not been investigated for CCS on code.

During training, the model is presented a batch of contexts and target blocks and learns to match each context with its corresponding target, as in typical contrastive learning (see Section 2.2.2). The goal is to learn an encoder that produces meaningful dense representations, called embeddings, for the code snippets that accurately capture semantic relevance scores for CCS retrieval. After training, the retriever can then be used for search, by first encoding all code snippets within the codebase with the model and storing them

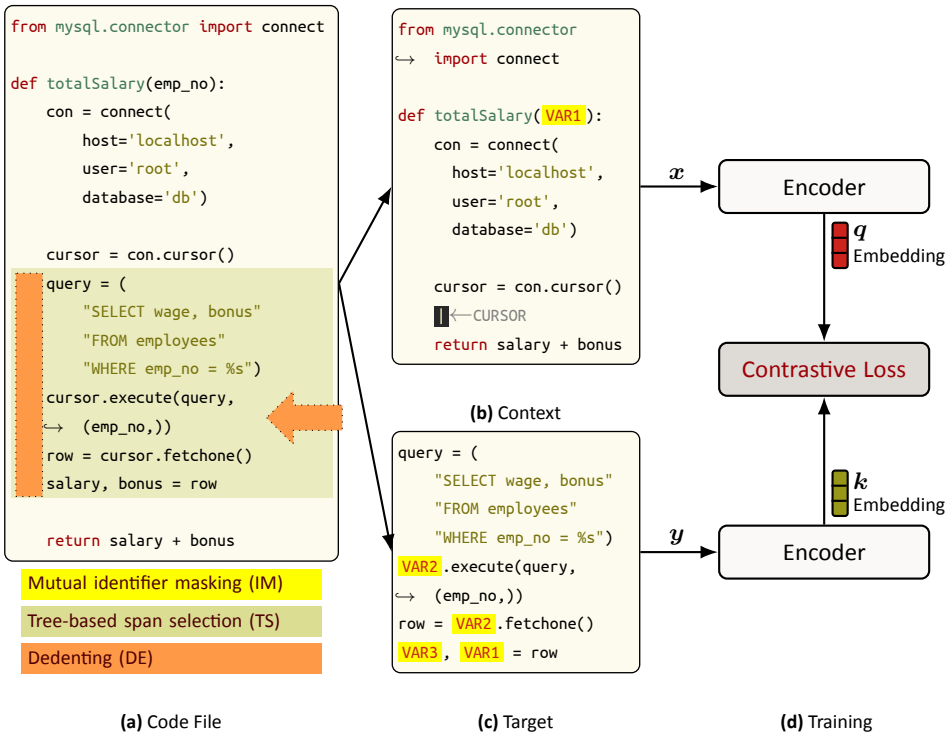


Figure 5.3: The proposed self-supervised learning process bootstraps context-target pairs (b) and (c) for contrastive learning (d) by randomly removing a target passage from a code file (a). To avoid leakages (see Figure 5.2) this chapter proposes to use the following deleaking steps: (TS) the target is selected using the code's syntactic structure (green) instead of using random spans. Additionally, (DE) the target is dedented (orange arrow), and (IM) mutual identifiers are masked either in the context or target (yellow).

in a vector database. At inference time, a code context (or query) is encoded, and the resulting embedding is matched against all embeddings in the database using nearest neighbor searches (see Section 2.2.2).

Leakage Patterns A severe key risk for the learning approach outlined above is that the encoder can exploit trivial patterns—or "*leakage*"—between context and target during pair matching and overfit on these patterns, without learning actual semantic similarity. This is visualized in Figure 5.2, where the following leakage patterns exist:

(P1) Context and target snippet share identifiers, whereas real solutions might come with different identifiers. See, for example the shared identifiers `con` and `connect` that are highlighted in yellow in Figures 5.2b and 5.2c.

(P2) A naive cut to create the context and target pair splits syntactic primitives, so that the pair can be identified by matching delimiters. This is visualized in red

in Figure 5.2, where the closing bracket to the one in the context is in the target snippet.

(P3) The target's indentation level always matches the one required by the context. This is visualized in orange in Figures 5.2b and 5.2c.

To address these patterns, this chapter's first contribution is a novel approach towards self-supervised CCS, that introduces deleaking steps during pair construction which remove the above leaking patterns. Hence, in order to create context-target pairs free of leakage, the following deleaking steps are proposed:

1. **Mutual identifier masking (IM)**: Hides mutual identifiers, i.e., identifiers that appear in both context and target, either in the context or the target snippet (yellow in Figures 5.3b and 5.3c), which addresses Leakage (P1).

2. **Tree-based span selection (TS)**: Selects the target snippet based on the code's syntactic primitives (green in Figure 5.3a), which addresses Leakage (P2).

3. **Dedenting (DE)**: Dedents the target to indentation level 0 (orange arrow), which addresses Leakage (P3).

The second key research problem is the evaluation of CCS models. In natural language code search large evaluation datasets can be automatically bootstrapped from function docstrings (Husain et al. 2019), or code clone detection, where evaluation datasets have been curated manually (Svajlenko and Roy 2015). To the best of the author's knowledge, no such datasets exist for CCS. Lu et al. (2022) and Parvez et al. (2021) have evaluated CCS code retrievers in combination with generators, which use the context and the retrieved passages for code infilling. Thus, the quality of the encoder is evaluated only indirectly via the quality of generated passages. This is problematic, since code generator quality measures such as CodeBLEU (Ren et al. 2020) are known to reflect semantic correctness poorly, and generation quality does not only depend on the degree to which semantically relevant passages are retrieved, but also on generator capacity and other hyperparameters. Overall, it is not validated properly if for a test case relevant passages are successfully retrieved. To address this research problem, another contribution of this chapter is a manually curated dataset for the zero-shot evaluation of CCS models based on aligned code clones. We call this dataset Cocos (COntextualized COde Search Dataset).

On Cocos this chapter demonstrates that the proposed cloze task-based self-supervised code retrieval training without the proposed deleaking steps performs worse than statistical baselines, such as BM25 (Robertson and Zaragoza 2009). When the deleaking steps are applied the retrieval quality is significantly improved, and the performance compared to statistical baselines more than doubled. A final contribution explores the possible benefits of this style of encoder pretraining on the code classification and similarity tasks *defect* and

code clone detection on CodeXGlue (Lu et al. 2021). Here, state-of-the-art results are achieved that even improve over the results of the structural pretraining from Chapter 4.

In summary, the key contributions of this chapter are:

1. Introduce the Cocos dataset, which is to the best of the author's knowledge the first evaluation dataset for Contextualized Code Search (CCS).
2. Present a novel self-supervised approach to CCS, that is to the best of the author's knowledge the first approach that studies Cloze Task (CT)-based contrastive pretraining on source code. Additionally, this chapter introduces novel deleaking steps that significantly improve the retrieval performance after pretraining, more than doubling the results of a commonly used statistical BM25-baseline.
3. Demonstrate that the presented self-supervised cloze task-based pretraining with deleaking steps achieves state-of-the-art performance for code retrievers, when evaluated on two encoder-based code understanding tasks from CodeXGlue: defect detection and code clone detection. For these two tasks the approach outperforms the previous state-of-the-art results from the last chapter (see Sections 4.5.7 and 4.6.4) by +1.1 p.p. accuracy and +1.8 p.p. MAP, respectively.

5.2 Related Work

Software reuse has been a long-studied topic in software engineering (Frakes and Nejmeh 1987; Fischer et al. 1991), largely because "it is often cost-effective to find similar applications that can be used as the basis for prototypes rather than building them from scratch" (Grechanik et al. 2007). According to Fischer et al. (1991), software reuse consists of three main phases: retrieval, comprehension, and adaptation. With code collaboration platforms like GitHub, code to reuse has become more accessible, yet finding relevant examples remains a challenge for developers, as they are often unaware of existing reusable solutions or when to use it (Ye and Fischer 2002). Thus, this chapter focuses on the retrieval phase of software reuse, where developers search for relevant code snippets in a codebase. Note that the main focus is on discovering small-scale code snippets, even though it is possible to search for prototypes of entire applications or complete products on GitHub for reuse.

This work is closely related to IR techniques used in NLP that often serve as inspiration for code search tasks. For related work about IR techniques in NLP see Section 2.5.

5.2.1 *Natural Language Code Search*

Earlier systems relied on heuristics to retrieve code snippets. JSearch (Sindhgatta 2006) extracts structural information from ASTs to enable specific queries, such as identifying a

class inheriting from a specific class. However, over the years, various systems have opted to instead use keyword queries for source code retrieval. Exemplar (Grechanik et al. 2007; Grechanik and Poshyvanyk 2008) and Sniff (Chatterjee et al. 2009) focused on API calls and application descriptions for retrieval. Bajracharya et al. (2010) improved Sourcerer's (Linstead et al. 2009) text-based and graph-based retrieval heuristics by incorporating common API usage patterns. CodeHow (Lv et al. 2015) expanded queries by matching them with API descriptions from the indexed codebase.

However, such heuristic-based approaches are limited by the quality of the heuristics used. Focus in code search has recently shifted to feature-less neural approaches, which have demonstrated superior performance and learn code search on large datasets of methods and their comments (Husain et al. 2019). Neural approaches enable the learning of semantic similarities, such as recognizing that `database` is similar to `db`, which keyword-based approaches often struggle to achieve. Thereby, code language models are typically first pretrained on unimodal code, but also bimodal code-comment data to learn the relationship between code and natural language, and subsequently fine-tuned with contrastive learning for code search. LMs that have been trained on bimodal data include CodeBert, GraphCodeBert (Guo et al. 2021), SynCobert, CodeRetriever (Li et al. 2022), ContraBERT (Liu et al. 2023), CoTexT (Phan et al. 2021), CodeT5, and UniXcoder (Guo et al. 2022). Just like the approach developed in this chapter these models are all transformer-based. For detailed descriptions of these models, refer to Section 4.2.

The bimodal pairs for natural language code search are bootstrapped from code files by using the AST to detect methods and their corresponding docstrings (Husain et al. 2019). Then they use only the first sentence of the docstring as the natural language query. They address two types of leakage: By design cutting-out code snippets eliminates syntactic leakage (P2), the same way it is done in this chapter. By using the first sentence of the docstring as the query, they reduce the leakage of identifiers (P1) (since after the preceding summary a docstring usually describes each parameter). However, compared to the work in this chapter, this is done in a preprocessing step and not dynamically during training.

5.2.2 *Self-Supervised Contrastive Learning for Code*

Inspired by trends in NLP several approaches have been developed that use contrastive learning during pretraining to improve the performance of code language models. Since contrastive learning requires positive pairs of semantically similar examples for training and annotated data is limited, various approaches have been developed to augment code snippets for this purpose. ContraCode (Jain et al. 2021) uses a compiler for semantically equivalent code transformations, such as code compression, identifier modification, and regularization, to form positive pairs. Similarly, Corder (Bui et al. 2021) augments

code snippets to create functionally equivalent versions, with variable renaming, dead code insertion, and syntactical transformation. The authors train a transformer model end-to-end on the resulting positive pairs. Disco (Ding et al. 2022) constructs not only positive pairs, but also creates buggy versions of code snippets to form negative pairs. Buggy versions are created by introducing syntactic and semantic errors, such as changing boolean operators or parameters to function calls. CodeRetriever (Li et al. 2022) trains on bimodal data and additionally mines noisy unimodal positive pairs, by comparing method names and docstrings using unsupervised SimCSE sentence embeddings (Gao et al. 2021b). The SimCSE method is also used by UniXcoder (Guo et al. 2022) which learns contrastive code representations by running the same code piece twice through a transformer with dropout. ContraBERT (Liu et al. 2023) augments not only code but also comments with back-translation and word manipulation. SynCobert uses contrastive learning on code-comment pairs, and—in addition to the bimodal training—detects differently masked and ordered input sequences of unimodal code. This eliminates the need for augmentations.

None of these approaches separates the code into a context and a target code snippet; instead, the same snippet is augmented to create multiple versions. This obviously leads to much more leakage and thus the augmentations need to be much more sophisticated. In contrast, the deleaking steps proposed in this chapter are much simpler. Additionally, the aforementioned augmentation strategies could be applied to the approach developed in this chapter as well. Augmentation alone does not address the structural leakage patterns (P2) and (P3) (see Section 5.1.1), between the context-target pairs. However, deleaking step mutual identifier masking can be also considered a form of augmentation—although much simpler than the ones used in the aforementioned approaches.

5.2.3 Contextualized Code Search

Contextualized Code Search (CCS) aims to retrieve complementary pieces of code given an incomplete context. Thereby, the system needs to infer the intended semantics of the missing piece by analyzing the current developer's context in order to retrieve relevant snippets. Some non-neural approaches have studied CCS scenarios, but mostly compute a context-context similarity, in order to find similar contexts which may contain additional functionality for reuse. CodeBroker (Ye and Fischer 2002) uses an incomplete method's docstring and signature for retrieval to recommend code snippets without the need for manual query formulation. Strathcona (Holmes and Murphy 2005; Holmes et al. 2005) generates structural contexts and matches these with heuristics, such as inheritance and API call similarity, to locate relevant examples. XSnippet (Sahavechaphan and Claypool 2006) extracts structural information from the current developer context to find relevant code snippets. Mishne et al. (2012) develop a semantic code search algorithm, that uses typestate information for answering API-usage queries in partial programs,

where multiple parts can be missing. FaCoY (Kim et al. 2018) matches the context with StackOverflow tags, extracts code snippets from posts with these tags, and uses them to find similar code in the codebase. Aroma (Luan et al. 2019) searches for code snippets that approximately contain the context code using manually constructed structural features and then intersects the results with the context to narrow the results. Since all these approaches are non-neural there is no need to address leakage.

Neural approaches include codec (Mukherjee et al. 2020), which learns CCS by decompiling functions into a simpler intermediate representation called sketch (Murali et al. 2018) and uses a neural network to find similar fragments. scotch (Dahal et al. 2022) enhances natural language code search with file context by fine-tuning CodeBert on a custom dataset to retrieve complete functions based on comments and file context. The authors do not address the leakage that results by extracting functions from code contexts. ReAcc (Lu et al. 2022) focuses on improving retrieval-augmented code completion. The authors train a code retriever by augmenting code snippets with dead code and variable renaming. Instead of matching partial target snippets with incomplete contexts like the approach in this thesis, the authors train retrieving full augmented files. Obviously, even with their leakage reduction steps, this is a much easier retrieval scenario.

It is important to note that no prior work, to the author's knowledge, has applied contrastive pretraining on source code using a cloze task-based objective for CCS. While the CCS task itself has been occasionally explored, it remains an under-researched area, particularly with regard to self-supervised training. The approach developed in this chapter is novel in bootstrapping the context-target pairs for contrastive learning, and additionally addresses leakage with simpler and more targeted techniques than previous work, that mostly augmented the same code snippet twice (which obviously requires more sophisticated leakage reduction steps). Also, CCS as a machine learning task is much more challenging, requiring the model to build better semantic representations.

5.3 Approach

In a retrieval scenario, a query $\boldsymbol{x}$ is used to retrieve a passage $\boldsymbol{y}$. In CCS, the query is a code context with a cursor position, and the passage or target is a useful snippet. Given $\boldsymbol{x}$, a retriever learns to find $\boldsymbol{y}$ among other passages, and is trained with a typical siamese contrastive learning setup (see Section 2.2.2). However, as outlined above sufficient annotated training data is scarce and hard to obtain. Hence, this chapter explores a self-supervised strategy towards CCS, that bootstraps the pair from an arbitrary piece of code with a Cloze Task (CT).

From an arbitrary code token sequence $\boldsymbol{c} = (c^{(1)}, \ldots, c^{(l)})$, a subsequence of length L is selected as a possible relevant target $\boldsymbol{y}$. To construct the context token sequence $\boldsymbol{x}$, the target $\boldsymbol{y}$ is replaced by a special mask token $x^{(mask)}$ that marks the cursor position. After

replacement, the sequences are:

$$\boldsymbol{x} = (c^{(1)}, \ldots, c^{(i-1)}, x^{(mask)}, c^{(i+L+1)}, \ldots, c^{(l)}) \tag{5.1}$$

$$\boldsymbol{y} = (c^{(i)}, \ldots, c^{(i+L)}) \tag{5.2}$$

Note that after creating $\boldsymbol{x}$ and $\boldsymbol{y}$, as in Section 4.4, a programming language-specific identifier token $x^{(lang)}$ is prepended to both sequences. This token has the same function as the CLS-token used in BERT, and can be used by the model to summarize the sequence or to point attention to when not needed (Clark et al. 2019). These sequences can now directly be used as an input to a contrastive architecture, as visualized in Figure 5.3d, and trained with a contrastive loss.

For the retrieval task, the model is trained to maximize the similarity between the context and target sequence, while minimizing the similarity between the context and negative samples (see Section 2.2.2). As detailed in Section 2.3, the transformer encoder outputs a sequence of hidden states $\boldsymbol{z}^{(i)} \in \mathbb{R}^d$ in its final layer. In the case of the SYNTAXPT model, d=768. This work follows recent approaches that use the embedding of the first token as a sequence embedding (Lee et al. 2019; Ren et al. 2021; Gao et al. 2021b). In our case, this corresponds to the embedding of the language identification token $\boldsymbol{z}^{(lang)}$. Given the embeddings of the context and target sequence $\boldsymbol{q}$ and $\boldsymbol{k}$, the cosine similarity between the two is computed[2]. The weights of the context and target encoder are shared, and initialized with pretrained weights from SYNTAXPT's encoder, The model is trained with the contrastive InfoNCE loss from Equation (2.19).

5.3.1 *Deleaking Steps*

The construction of the context-target pair from a single piece of code with a cloze task is self-supervised and inherently addresses Challenges (1) and (3) (see Section 5.1.1), as the model is explicitly trained to retrieve snippets similar to the missing piece. Unfortunately, the cut-out code segments make poor proxies for real-world queries due to the leakage patterns (again Section 5.1.1). A retriever can exploit these patterns and overfit to the synthetic data without learning actual semantic retrieval. To address these patterns, this work proposes three novel deleaking steps: tree-based span selection (TS), mutual identifier masking (IM), and dedenting (DE). The steps can be applied individually from each other, but also all together to $\boldsymbol{x}$ and $\boldsymbol{y}$. Tree-based span selection controls the selection of $\boldsymbol{y}$ from $\boldsymbol{c}$, while mutual identifier masking and dedenting post-process the pair after selection.

[2]While other similarity or distance metrics, such as L2, can also be used, cosine similarity offers the advantage of being computable through a scalar or matrix product after normalizing the embeddings. This can be advantageous at retrieval time, e.g., to quickly compute the similarity between one or multiple queries and stored passages.

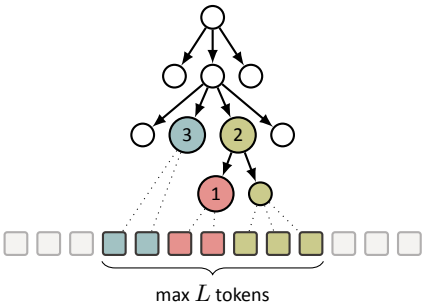


Figure 5.4: The tree-based span selection technique selects a target $\boldsymbol{y}$ with a maximum length of L tokens from the syntax tree of a code snippet. First the span length L and a node (1, red) are sampled. Then the selection is iteratively expanded to the parent (2, green), and a sibling (3, blue) until the target has the desired length or no further expansion is possible.

Tree-based span selection (TS)

Naively selecting a random token-level subsequence as a target is prone to leakage as outlined in Figure 5.2. The naive selection highlighted in red cuts several syntactic constructs, and leaves several brackets unmatched. To this end, this chapter proposes to use the tree-based span selection technique from Section 4.4.1 to define $\boldsymbol{y}$ based on the program's syntax tree. This step is abbreviated throughout this chapter as tree-based span selection (TS). This technique always prevents structural leakage (P2), as the selected pieces of code are syntactically complete by definition.

In this chapter, the subtree sampling process differs slightly from the approach introduced in Section 4.4.1 and is visualized in Figure 5.4. Here, first the length L of $\boldsymbol{y}$ is sampled from a Gaussian distribution $\mathcal{N}(150, 90^2)$. Then a node N which covers at most L tokens is sampled uniformly from the tree's nodes. This node may not span over all L tokens, but less, so the selection is iteratively expanded to either N's parent or one of N's direct siblings until approximately L tokens are selected. When N's parent spans over at most L tokens, the parent is used and the process repeated. Otherwise, the selection is iteratively expanded to one or more of N's direct siblings (if possible). This step is important, as it enables producing multi-statement targets that span over multiple lines, such as the green selection in Figure 5.3a, since most often each line is a separate expression node in the tree with the same parent.

Using the syntax tree could additionally enable the restriction of targets to meaningful code primitives such as functions or loops, however, this is left to future work. Figure 5.6 shows multiple targets produced by applying tree-based span selection to the code context from Figure 5.3a. Note that these examples will be discussed in detail in Section 5.3.2.

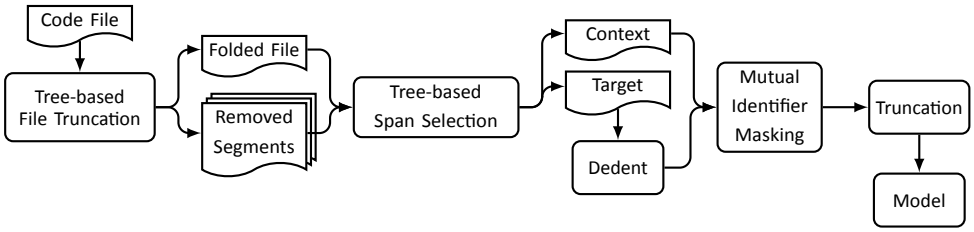


Figure 5.5: The training pipeline for CCS.

Mutual identifier masking (IM)

At inference time, relevant solutions may come with different identifiers than the context (Challenge (2)). Obviously, this is not the case when constructing the pair from the same code snippet. The model can exploit matching identifiers between context and target, so the pair matching during contrastive training gets easy, without learning actual semantic similarity. Hence, identifiers in $\boldsymbol{x}$ or $\boldsymbol{y}$ are randomly masked by special tokens, such as `VAR1` or `VAR2`. This is visualized in yellow in Figures 5.3b and 5.3c. However, Lee et al. (2019) found that word matching is a useful feature for retrievers, who can exploit the lexical overlap.

To provide as much lexical information as possible for the model, only mutual identifiers are masked, i.e., ones that appear in both context and target code. Specifically, from the set of mutual identifiers only 90% will be masked. This keeps a little bit of lexical overlap and the model cannot exploit the fact, that an identifier can never appear in both context and target. The set of mutual identifiers is randomly partitioned to be masked either in the context or in the target snippet. This ensures the model cannot exploit the amount of masked variables in either context or target. For generalization and to avoid overfitting by analyzing the hidden mutual identifiers, also 10% of the not mutual identifiers are masked. Following Lee et al. (2019), for 5% of samples mutual identifier masking is skipped altogether, so that the model can learn to exploit word matching as a feature.

Dedenting (DE)

A relevant solution could have any type of indentation level. However, when constructing a context-target pair from a single code segment, the pair will always have the same indentation level. This can be another clue for the model to exploit and overfit. Hence, the indentation of $\boldsymbol{y}$ is removed in 90% of the training samples. To do so, the indentation level or the amount of spaces or tabs before the mask token $x^{(mask)}$ is determined and the same amount of spaces removed from the beginning of every line in $\boldsymbol{y}$.

5.3.2 Training Pipeline

The training pipeline visualized in Figure 5.5 is implemented dynamically. To this end, the pipeline uses the TENSORTREE library introduced in Section 4.5.4 on trees of pre-tokenized code files. It begins by truncating regular, possibly too large code files using the tree-based file truncation technique described in Section 4.4.2. This step outputs smaller code snippets ranging from 150 to 800 tokens. Note the sequences are still longer than the model's maximum size of 512 tokens, but will be shortened in subsequent steps of the pipeline (e.g., by selecting the target). Tree-based file truncation produces two kinds of outputs: (1) a context-rich folded file, in which fold-tokens indicate that at these positions some nodes have been removed, along with (2) the removed code segments as individual samples. Following this, the tree-based span selection step splits the single code segment into a context-target pair, as previously described. Due to the probabilistic implementation, this step may produce a pair where the context or target exceeds 512 tokens (recall that segments up to 800 tokens were selected). Therefore, after applying the remaining deleaking steps to the pair, the final sequences are truncated token-wise to a maximum of 512 tokens before being fed into the model. During the token-wise truncation of the context, the mask token is ensured to remain within the context.

Note that most steps operate probabilistically, so that each epoch exposes the model to new examples. This is illustrated in Figure 5.6, where three different context-target pairs are shown, generated by the training pipeline based on the code context from Figure 5.3a. The highlighted lines denote the code segment sampled by the tree-based span selection technique and the corresponding context-target pair to which unique identifier masking and dedenting have been applied. It can be seen that all bootstrapped pairs are useful, in a way that they represent actual information needs that a developer might have. For example, in the first iteration, the functionality connecting to a database was hidden, while in Figure 5.6i, the SQL query was hidden. Both target snippets resemble information that an inexperienced developer might find helpful during coding, for instance if the developer does not know how to formulate SQL.

Several notable aspects of the training pipeline are worth discussing. In all iterations, the technique produced challenging context-target pairs. For instance, in the context shown in Figure 5.6b, three identifiers were hidden, whereas in the target in Figure 5.6c, only one identifier was hidden. This randomized number of hidden identifiers makes it more difficult for the model to exploit the quantity of masked variables. Additionally, in both the context and target, `VAR1` masked a non-mutual identifier (`row` and `connect`), due to the general mask probability, with the identifier `row` hidden in line 4 but not in line 5. In Iteration 1, the tree-based span selection produced a smaller context than target, this ensures the model can handle varying lengths of context and target snippets at inference time. While the first two iterations selected top-level statements within the method, in

```
def totalSalary(emp_no):
  con = connect(
    host='localhost',
    user='root',
    database='db')
  cursor = con.cursor()
  query = (
    "SELECT wage, bonus"
    "FROM employees"
    "WHERE emp_no = %s")
  cursor.execute(query, (emp_no,))
  row = cursor.fetchone()
  salary, bonus = row
  return salary + bonus
```

(a) Iteration 1: Selection

```
def totalSalary(emp_no):
  |<—CURSOR
  VAR2.execute(VAR3, (emp_no,))
  VAR1 = VAR2.fetchone()
  salary, bonus = row
  return salary + bonus
```

(b) Iteration 1: Context

```
con = VAR1(
  host='localhost',
  user='root',
  database='db')
cursor = con.cursor()
query = (
  "SELECT wage, bonus"
  "FROM employees"
  "WHERE emp_no = %s")
```

(c) Iteration 1: Target

```
def totalSalary(emp_no):
  con = connect(
    host='localhost',
    user='root',
    database='db')
  cursor = con.cursor()
  query = (
    "SELECT wage, bonus"
    "FROM employees"
    "WHERE emp_no = %s")
  cursor.execute(query, (emp_no,))
  row = cursor.fetchone()
  salary, bonus = row
  return salary + bonus
```

(d) Iteration 2: Selection

```
def VAR1(emp_no):
  con = connect(
    host='localhost',
    user='root',
    database='db')
  cursor = con.cursor()
  query = (
    "SELECT wage, bonus"
    "FROM employees"
    "WHERE emp_no = %s")
  cursor.execute(query, (emp_no,))
  |<—CURSOR
```

(e) Iteration 2: Context

```
row = VAR1.fetchone()
salary, bonus = row
return salary + bonus
```

(f) Iteration 2: Target

```
def totalSalary(emp_no):
  con = connect(
    host='localhost',
    user='root',
    database='db')
  cursor = con.cursor()
  query = (
    "SELECT wage, bonus"
    "FROM employees"
    "WHERE emp_no = %s"
  )
  cursor.execute(query, (emp_no,))
  row = cursor.fetchone()
  salary, bonus = row
  return salary + bonus
```

(g) Iteration 3: Selection

```
def totalSalary(emp_no):
  con = connect(
    host='localhost',
    user='root',
    database='db')
  cursor = con.cursor()
  query = (
    |<—CURSOR
  )
  cursor.execute(query, (emp_no,))
  row = cursor.fetchone()
  salary, bonus = row
  return salary + bonus
```

(h) Iteration 3: Context

```
"SELECT wage, bonus"
"FROM employees"
"WHERE emp_no = %s"
```

(i) Iteration 3: Target

Figure 5.6: Three different context-target pairs created from the code context from Figure 5.3a by the implementation of the training pipeline. Highlighted lines denote the code segment sampled by the tree-based span selection technique. Mutual identifier masking and dedenting have been applied to the context-target pair.

Iteration 3, as shown in Figure 5.6g, a deeper multiline string was sampled. However, since a string contains no identifiers, no mutual identifier masking was applied.

5.4 Evaluation Dataset for Contextualized Code Search

In CCS, the objective is to find code segments that implement the intended functionality for a given code context query with missing parts. Quantitative evaluation of CCS would need a dataset with relevancy assessments. Segments that implement the intended functionality are considered relevant results. Ideally, for an accurate evaluation, for a query multiple relevancy annotations should be available. While query contexts can be easily created by removing blocks of code, obtaining suitable blocks that implement the same functionality as the removed part is challenging. To the best of the author's knowledge, datasets for CCS are not available.

Send E-Mail	20	Encryption Key Files	18	Download From Web	15
CRC32 File Checksum	23	Play Sound	22	Decompress zip archive	7
Connect to Database	27	Take Screenshot to File	22	Bubble Sort Array	22
Get MAC Address String	16	Fibonacci	20	Setup SGV	18
Delete Folder and Contents	18	Encrypt To File	21	Setup SGV Event Handler	11
Parse CSV File	22	Open URL in Browser	24	Initialize Eclipse Project	18
Transpose a Matrix	20	Open File in Desktop	23	Get Prime Factors	18
Extract Matches Using Regex	20	GCD	20	Shuffle Array in Place	20
Copy Directory	21	Convert Date String	20	Binary Search	21
Test Palindrome	18	Zip Files	20	Load Custom Font	21
Write PDF File	20				

Table 5.1: The number of samples for each functionality in Cocos.

Thus, this section introduces the Cocos dataset for CCS based on BigCloneBench, a Java code clone benchmark with various types of annotated clones (Svajlenko and Roy 2015). Particularly BigCloneBench includes Type IV clones, which are defined by functional similarity as "two or more code fragments that perform the same computation but are implemented through different syntactic variants" (Roy and Cordy 2007). The Type IV clones in BigCloneBench are annotated at a method level to implement specific functionalities. For instance, Figures 5.7a and 5.7c show two Java functions that implement the functionality *Compute a CRC32 checksum for a file*.

The clones from BigCloneBench serve as ideal proxies for real-world queries and solutions for CCS. To construct the CCS dataset, functions from 31 randomly selected functionalities were considered, as shown in Table 5.1. However, the functionality may constitute only a portion of each function. To create realistic information needs, annotators manually identified a shared "main functionality" between the clones. When the sub-block is removed, a code context is created, and the corresponding sub-blocks from the other functions are deemed relevant search results.

```java
172 public int calculatePrimaryEntriesChecksum() {
173     CRC32 checksum = new CRC32();
174     for (GPTEntry entry : entries)
    ↪   checksum.update(entry.getBytes());
175     return (int) (checksum.getValue() & 0xFFFFFFFF);
176 }
```

(a) A function that computes the CRC for a file.

```java
1 public int calculatePrimaryEntriesChecksum() {
2     CRC32 checksum = new CRC32();
3     |  ←CURSOR
4 }
```

(b) The query obtained by removing the highlighted part.

```java
405 public static long getCRC(File f) throws IOException {
406     InputStream is = new FileInputStream(f);
407     CRC32 crc = new CRC32();
408     byte[] data = new byte[1024];
409     int read;
410     while ((read = is.read(data)) > -1) {
411         crc.update(data, 0, read);
412     }
413     return crc.getValue();
414 }
```

(c) A function with the same functionality (highlighted).

```java
1  HashSet<Integer> lst = new HashSet<Integer>();
2  for (int i = 0; i < values.length; i++) {
3      lst.add(values[i]);
4  }
5  int[] v = new int[lst.size()];
6  Iterator<Integer> it = lst.iterator();
7  for (int i = 0; i < v.length; i++) {
8      v[i] = it.next();
9  }
10 return v;
```

(d) A distractor snippet.

Figure 5.7: Two Java functions (a) and (c) that compute the CRC32 checksum of a file have been annotated in BigCloneBench to implement a certain functionality. However, this functionality may constitute only a portion of each function. During the construction of Cocos, the highlighted sub-blocks (blue in (a) and (c)) were manually annotated to contain the same functionality. Subsequently, when the highlighted part is removed in (b) a code context is created, for which the corresponding highlighted lines from the other function (c) are relevant search results. The last sub-figure, (d) shows an irrelevant distractor target from the Cocos dataset.

For instance, the highlighted sub-blocks (blue in Figures 5.7a and 5.7c) were manually annotated to contain the same functionality. Figure 5.7b shows the code context for which the highlighted lines in Figure 5.7c are expected to be found. In Figures 5.7a and 5.7c, the initialization of the `CRC32` object (Lines 173 and 407) is also shared functionality, but the annotator used their discretion to create a realistic scenario in which the developer might know which objects to use, but not how to add the content of the file to create the checksum. Note that the query and the relevant solution have a low amount of overlapping identifiers. For instance, the only overlapping identifier is `CRC32` in the query in Figure 5.7b and `crc` in the blue part of the possible solution in Figure 5.7c.

This process yielded a total of 606 context-target pairs, on average approximately 20 pairs per functionality. To simulate a more realistic search index, 10,000 distractor snippets were created from Java functions in CodeSearchNet (Husain et al. 2019) by randomly selecting top-level statements from the functions' bodies. An example distractor snippet is visualized in Figure 5.7d. Now for each context, such as the one in Figure 5.7b, the performance of models in retrieving targets that implement the same functionality can be quantitatively measured. Note that the original removed segment is not considered during evaluation, given that it has the same identifiers. Additional examples are shown in Figure A.5 in the Appendix.

5.4.1 Evaluation Protocol: Zero-shot Code Retrieval

On Cocos, models are evaluated in a zero-shot setting. The scale of the dataset allows no fine-tuning. For each context, all possible targets and the 10,000 distractor snippets are ranked, while the original target is excluded. No cut-off is applied, so all targets are considered. To evaluate the model, standard information retrieval metrics Prec@k, MAP, and nDCG are used.

5.5 Experimental Setup

The experiments aim to assess whether the proposed approach for self-supervised CCS is effective. To this end, the Cocos dataset was created. On this dataset, the impact of each deleaking step is investivated and the approach compared to other information retrieval baselines. Only unsupervised baselines are considered for the Cocos dataset, as no labeled data is available. Additionally, this chapter explores whether this type of pretraining improves the code-encoder quality of the transformer model, which could potentially make it useful as a general pretraining task for various models, even outside the context of CCS.

5.5.1 Research Questions

The following research questions are addressed through the experiments:

Research Question 5.1: *Can CCS be learned with a self-supervised Cloze Task (CT)?*

This research question investigates to what extent a model trained using a self-supervised cloze task retrieval approach on source code can perform CCS on the Cocos dataset. The cloze task may not be ideal for code due to context-target pair leakage, as discussed previously. If leakage occurs, it could substantially impact the model's ability to learn semantic similarity, reducing its performance on Cocos. To assess the impact of leakage, multiple ablation models were trained with and without the different deleaking steps, and their performance on Cocos was evaluated. When tree-based span selection is omitted, a random token-level span of the same length distribution is selected as a replacement. The remaining deleaking steps are simply skipped.

Research Question 5.2: *How does the self-supervised CCS approach compare to statistical baselines such as BM25 (Robertson and Zaragoza 2009)?*

Statistical retrieval systems, such as BM25, have become the industry standard. However, these systems rely on keyword matching, which is suboptimal for source code as semantically identical solutions might come with different identifiers. Thus, the proposed CCS model is compared to statistical baselines using the Cocos dataset.

Research Question 5.3: *Does pretraining with the self-supervised CCS approach improve the code encoder's performance on other code understanding tasks?*

The proposed pretraining strategy for CCS could improve not only the model's ability to perform CCS, but also the model's general code understanding capabilities. Overall, retrieving complementary pieces of code is a challenging task, that requires the model to semantically understand the snippets. To evaluate this, the pretrained model is tested on benchmark datasets from CodeXGlue on two code understanding tasks. Specifically, the tasks defect detection and code clone detection are chosen from CodeXGlue, since these tasks are encoder-based tasks and have been part of the evaluation from the last chapter. Which allows comparing the performance to state-of-the-art models and SyntaxPT.

5.5.2 Hyperparameters and Setup

Chapter 4 has demonstrated that the SyntaxPT model achieves state-of-the-art code understanding capabilities, even when evaluated on encoder-based tasks from CodeXGlue. Hence, unless otherwise stated, all models are initialized with the weights of the encoder of SyntaxPT, to give the model a head start in code understanding capabilities. This work builds on the implementation of SyntaxPT and thus uses a similar experimental setup and infrastructure. For additional details please refer to Chapter 4.

For training, the large-scale dataset introduced in Section 4.5.5 is also used to train the retriever model. All tasks and models use the AdamW optimizer (Loshchilov and Hutter 2019) with a learning rate schedule that warms up the learning rate linearly for 10% of the training steps, along with a polynomial decay for the remaining steps. The self-supervised contrastive model is trained for 500k steps on a single A6000 GPU, with a peak learning rate of 0.00005 and the dynamic batch size set so that batches contain around 7000 tokens. During training retrieval performance is measured on 30,000 held-out validation samples of the pretraining dataset. For final test results the checkpoint with the highest validation MRR is used.

Recall that the dataset contains code from multiple programming languages. This has the drawback that mixed batches with samples from different languages are another leakage pattern: Using negative samples from another programming language in the contrastive loss computation, does not require precise semantic understanding of the code. Those are easy to differentiate by the model. To address this issue, this work creates batches only with samples from one programming language. At the end of the training pipeline, the samples are first grouped by programming language, and subsequently formed to batches. Obviously, this creates more challenging batches than simply shuffling samples.

Model Features	MAP	NDCG	P@1	P@3	P@10
BM25 standard	12.4	43.8	27.9	24.9	17.1
BM25 camel	28.0	57.1	39.4	37.1	33.2
None	15.7	49.9	45.9	38.0	24.8
TS	26.5	59.6	58.1	50.8	37.0
TS, IM	33.8	66.0	69.8	61.0	45.3
TS, DE	36.3	65.9	59.4	54.6	44.4
TS, IM, DE	**50.9**	**76.3**	**73.6**	**70.3**	**59.7**

Table 5.2: Zero-shot code retrieval results for different deleaking steps as described in Section 5.3: tree-based span selection (TS); mutual identifier masking (IM); dedenting (DE). The non-neural BM25 (Jones et al. 2000) is compared once with the ElasticSearch default tokenizer (standard) and a code-specific tokenizer that splits on camel case (camel).

5.6 Results

This section presents the results of the experiments.

5.6.1 *Performance of Self-Supervised Contextualized Code Search*

Can CCS be learned with a self-supervised Cloze Task (CT)? – RQ 5.1

The results are summarized in Table 5.2. The model trained without any deleaking steps performs poorly on Cocos, likely because it retrieves mostly targets with matching identifiers rather than semantically similar code snippets. This is indicated by a high precision at 1 (P@1) but a low MAP score. The model fails to consistently retrieve *all* relevant targets. These results suggest that training a cloze task-based retriever without deleaking is not advisable. The performance on Cocos improves consistently when the deleaking steps are applied, and the combination of all deleaking steps achieves the best performance. The model with all deleaking steps will be called SyntaxPT-ccs from now on. SyntaxPT-ccs achieves a more than three times higher performance, compared to the model without deleaking steps (50.9% vs. 15.7% MAP). These results support the hypothesis that leakage hinders the learning process. Therefore, for effective code retrieval training using a cloze task, it is crucial to carefully select spans without syntactic leakage, handle mutual identifiers to enable the retrieval targets with different variable names, and also dedent targets appropriately.

This improvement is further illustrated in Figure 5.8, which presents a t-SNE visualization of the context and target embeddings from a random selection of Cocos problems. The model trained without deleaking steps fails to fully disambiguate contexts and targets from different problems. In contrast, the model trained with deleaking steps forms more distinct clusters in the embedding space. These observations lead to the conclusion that a cloze task-based code retriever without deleaking steps is not a viable option and the use of deleaking techniques is essential for performance optimization.

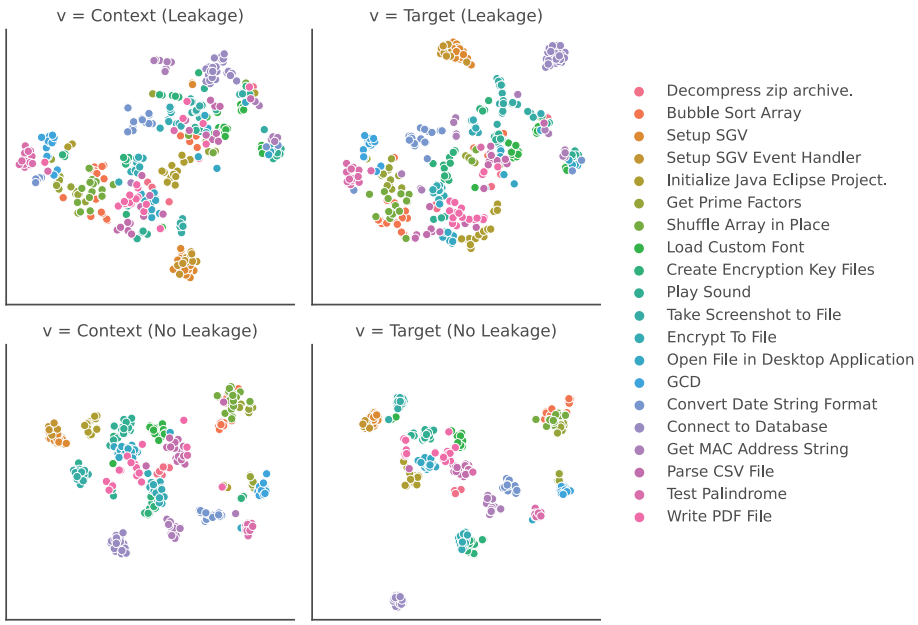


Figure 5.8: T-SNE visualization comparing the embeddings of models trained with (bottom) and without (top) deleaking steps. The embeddings of the model trained with deleaking steps show more distinct and well-formed clusters, which indicates an improved semantic understanding.

5.6.2 *Comparison to Statistical Baselines*

> *How does the self-supervised CCS approach compare to statistical baselines such as BM25 (Robertson and Zaragoza 2009)?* – RQ 5.2

Classic statistical retrieval systems, such as the Okapi BM25 (Jones et al. 2000) used in ElasticSearch, are widely employed in the industry and therefore used as a realistic baseline for comparison. These statistical methods primarily rely on keyword matching instead of semantic understanding, which is suboptimal for code since variable and class names might be different without differing semantics. This is intensified by source code naming conventions that often use camel or snake case word compositions, which—without proper tokenization—lead to even less lexical overlap. For example, the function names `calculatePrimaryEntriesChecksum` and `calculatePrimaryEntriesCRC` are two different tokens when using the standard ElasticSearch tokenizer, that keeps them as a whole. Therefore, in this approach the default tokenizer is compared to one that splits on camel and snake case. It outputs the tokens `calculate`, `Primary`, `Entries`, `Checksum` and `calculate`, `Primary`, `Entries`, `CRC` for the example. An overlap of three of four tokens more accurately reflects the similarity of the two identifiers. However, as argued above the method will fail when no lexical overlap is present, e.g., `database` and `db`.

Results are presented in Table 5.2. While BM25 models can retrieve some relevant targets, their performance is significantly lower compared to the proposed model. The models achieve 12.4% and 28.0% MAP with standard and camel case tokenization, respectively, versus 50.9% MAP for the proposed model. Interestingly, the BM25 baselines have lower P@1 and precision at 3 (P@3) scores, than all the neural models. This indicates, that the statistical baselines are not able to accurately differentiate the codes' semantics and thus retrieve relevant solutions rather on the lower ranks. One could think of using a multi-stage ranking approach, which is often used in IR, in which a keyword-based first-stage ranker retrieves a large amount of candidates (in the thousands), that are then reranked by a neural model in a second stage. This is left for future work.

5.6.3 General Encoder Quality

Does pretraining with the self-supervised CCS approach improve the code encoder's performance on other code understanding tasks? – RQ 5.3

The general quality of the encoder is assessed through evaluation on downstream tasks that require code understanding. In this chapter the defect and code clone detection tasks from CodeXGlue (Lu et al. 2021) are used, since they have been used to evaluate SyntaxPT in the last chapter and are encoder-only tasks that require no decoder for generation. Both tasks are described in detail in Section 4.5.7. The defect detection task is a binary classification task that predicts if a function contains a vulnerability. The clone detection task is a retrieval task that aims to retrieve semantically similar programs of the same functionality. This task is closely related to CCS, with the difference, that it compares complete code snippets with one another instead of partial code contexts with relevant targets. For both tasks a hyperparameter sweep is conducted, with the same setup as described in the aforementioned sections. The model checkpoint with the best validation performance is evaluated on the test set.

Results

The model trained with the self-supervised CCS objective outperforms all state-of-the-art models at the time of the experiments[3] by a large margin on both tasks, as shown in Table 5.3. For the clone detection task the previous state-of-the-art, the UniXcoder model, is outperformed by 0.8 p.p. MAP@500. On defect detection the model achieves the same performance as the best model on the CodeXGlue leaderboard at the time of writing in 2024 and outperforms all other models by over 2.7 p.p. accuracy.

[g] Results belong to ContraBERT model proposed by Liu et al. (2023).
[h] Results belong to UniXcoder model proposed by Guo et al. (2022).

[3] The experiments to this work have been done in 2022 and models that have been published up until December 2022 are included in this comparison.

When comparing SyntaxPT-ccs to the SyntaxPT one can see that on both tasks the performance improves when after structural generative pretraining the encoder is subsequently pretrained for CCS, before it is fine-tuned on the respective task. Especially on the clone detection task an improvement of 1.8 p.p. MAP could be observed. Obviously, a retrieval task benefits from a retrieval-style pretraining.

Model	Clone	Defect
	MAP@R	Accuracy
RoBERTa-code	76.7	61.1
CodeBert	82.7	62.1
Code2vec	-	62.5
Aroma	55.1	62.5
PLBart	-	63.2
GraphCodeBert	85.2	63.2
Disco	82.8	64.4
SynCobert	88.2	64.5
CodeRetriever	88.9	-
CodeT5-base	88.7	65.8
CoTexT	-	66.6
UniXcoder	90.5	-
2024 Leaderboard	90.5[g]	69.3[h]
SyntaxPT	89.5	68.2
SyntaxPT-ccs	**91.3**	**69.3**

Table 5.3: Results of the code clone and defect detection tasks on CodeXGlue. For task and dataset descriptions please refer to Section 4.5.7.

But the SyntaxPT-ccs model improves also on the classification task defect detection by 1.1 p.p. accuracy is achieved on the defect detection task. This shows that this style of pretraining leads to a strong general semantic understanding of code, not only useful for retrieval, but also for other encoder-based code understanding tasks. These findings indicate that the proposed CCS pretraining could be added to many pretraining approaches to improve their performance on code understanding tasks.

5.6.4 Comparison with OpenAI

In a last experiment new for this thesis, the SyntaxPT-ccs model is compared to OpenAI's commercial text and code embeddings on Cocos. These are OpenAI's flagship embedding models in September 2024, and are commonly used in Retrieval-Augmented Generation (RAG) applications. Note that the models are proprietary, for which the architecture, training data and algorithm are not publicly disclosed. Two models are available: `text-embedding-3-small` and `text-embedding-3-large`. Even though the models support adaptive sized output embeddings (Kusupati et al. 2022), this experiment uses the largest option for best possible performance. In this case, `text-embedding-3-small` outputs embeddings of size 1536, while `text-embedding-3-large` outputs embeddings of size 3072. In contrast, SyntaxPT-ccs has an embedding-size of 768. Additionally, one can expect that the OpenAI models have more parameters than the SyntaxPT-ccs model and are trained on both unsupervised and supervised data, but this is a speculation and not confirmed[4]. The cursor position—for which the target should be retrieved—is indicated to the models by a comment `// insert here` at the corresponding position in the context, since no markers (sentinel tokens) can be used in the OpenAI models.

[4]The only available research paper by OpenAI employees on text and code embeddings is Neelakantan et al. (2022). It is unclear if the available models (OpenAI 2024b) are the same as the ones used in the paper.

Model	Params	Dim	MAP	NDCG	P@1	P@3	P@10
SyntaxPT-ccs	110M	768	50.9	76.3	**73.6**	**70.3**	59.7
`text-embedding-3-small` (OpenAI 2024b)	?	1536	50.5	73.3	59.4	58.9	53.8
`text-embedding-3-large` (OpenAI 2024b)	?	3072	**60.0**	**78.8**	66.3	65.1	**62.5**

Table 5.4: Comparison of the SyntaxPT-ccs model to OpenAI's general purpose text and code embeddings on Cocos. These are proprietary models from OpenAI, for which the architecture, training data and algorithm are not disclosed.

Table 5.4 presents the results. One can see that the SyntaxPT-ccs model performs competitive with these commercial models. SyntaxPT-ccs performs better than OpenAI's smaller embedding model across all metrics, even though its embeddings are only half the size. Additionally, it performs better than `text-embedding-3-large` in P@1 and P@3, but worse in MAP and nDCG. This indicates that SyntaxPT-ccs peforms well in the top ranks, but does not have the same coverage as the larger OpenAI model. The `text-embedding-3-large` model achieves a MAP of 60.0% and a nDCG of 78.8%, which is an improvement of 9.1 p.p. in MAP and 2.5 p.p. in nDCG compared to the SyntaxPT-ccs model. The smaller improvement on nDCG confirms that the model is not as precise on the first elements in the result list, since nDCG penalizes errors in the beginning of the result list stronger. This is a promising result, since lower ranks are most important for a user. The better performance on the top-ranks can be attributed to the proposed training routine, which is specialized for CCS. Our model is trained to use the available information for which position a solution should be retrieved, which is not the case for the OpenAI models.

As these results demonstrate, our CCS model is competitive with OpenAI's general purpose embeddings, which have been developed, tuned and optimized by a much larger team of AI specialists, have likely been trained including supervised data (note that SyntaxPT-ccs's training is entirely self-supervised), and likely feature much bigger models.

5.7 Conclusion and Future Work

This chapter addressed the challenge of training a code retrieval model for Contextualized Code Search (CCS) without supervised data. To this end, a novel self-supervised approach was proposed, that uses a Cloze Task (CT) to bootstrap query-result pairs from unlabeled code for contrastive learning. To mitigate leakage patterns that could hinder the learning of semantic similarity, three deleaking steps—tree-based span selection, mutual identifier masking, and dedenting—were introduced. These steps effectively prevent the model from exploiting trivial patterns during training, forcing it to learn semantically meaningful code representations. Also, the Cocos dataset, the first evaluation dataset specifically designed

The smallest model size Neelakantan et al. (2022) study is 300M parameters, which is almost three times of SyntaxPT-ccs (110M). Their largest model has 175B parameters (1590 times larger).

for CCS based on aligned code clones, was introduced. This dataset enabled a rigorous evaluation of the retrieval capabilities of the model in a zero-shot setting.

The experimental results demonstrated that, without the proposed deleaking steps, cloze task-based self-supervised training underperforms compared to statistical baselines such as BM25. However, with the deleaking steps applied, the model significantly outperformed these baselines, and doubled the retrieval performance on the Cocos dataset. This confirms the effectiveness of the deleaking strategies in enhancing the model's ability to learn semantic code representations. Furthermore, it was shown that the self-supervised cloze task-based pretraining not only improves retrieval performance but also enhances general code understanding capabilities. On two encoder-based code understanding tasks from CodeXglue—defect detection and code clone detection—the model achieved state-of-the-art results and outperformed previous models, including the SyntaxPT model presented in the previous chapter.

These findings have important implications. They demonstrate that the proposed self-supervised cloze task-based pretraining with appropriate deleaking steps is a viable and effective strategy for CCS. It allows building powerful code retrieval models without the need for labeled data. Moreover, this approach contributes to the broader field of code understanding by offering a method to pretrain code-encoders that can perform well across various tasks. Future work could further train the proposed retriever in combination with a generator model for a joint, unsupervised RAG training (Izacard et al. 2020), where a generator and retriever model are trained at the same time, so that the retriever adds relevant code to the generator (e.g., from the project context).

An interesting direction for future work would be to integrate momentum contrastive learning (He et al. 2020) into the training routine, which aims to increase the amount of negative samples by keeping a queue of embeddings across batches, together with a moving-average of the encoder. He et al. (2020) show that this improves the quality of the learned representations on visual tasks, but recently has been shown to also work for NLP (Izacard et al. 2022). Larger amounts of negative samples require more sophisticated code understanding during pair matching in the contrastive loss computation. In combination with larger transformer models, this could improve the performance of the approach proposed in this chapter substantially. Also, it is left to future work to investigate, if the retrieval performance can be improved by adding some supervised data to the training routine, and how much is needed. The output size of the embeddings could also be investigated, which could potentially reduce the computational cost of the model.

Furthermore, the combination of this model with a multi-stage ranking approach commonly used in IR could be an interesting direction for future work. The SyntaxPT-ccs model could be used as a first-stage ranker to retrieve a large amount of candidates, which are then reranked by a cross-encoder, that can better differentiate the semantics of the code

snippets, but is computationally more expensive. This could potentially further improve the retrieval performance of the model. It is left for future work to investigate whether a cross-encoder can be trained with the same self-supervised cloze task approach, or if supervised data is needed for training. Another direction for future work could be to investigate the impact of training on mixed-language batches. The authors noticed that, since the model was trained exclusively on batches of a single programming language (see Section 5.5.2), the model is able to do cross-language retrieval (e.g., retrieve a Python solution for a Java problem).

Part II

Applications

The half of knowledge is to know where to find knowledge.

— Inscription at Dodd Hall, FSU

6

Evaluating Contextualized Code Search in Practical User Studies

The preceding chapters in Part I introduced several code-specific transformer models. Initially, two separate methods for incorporating the syntactical aspects of code into the transformer model were studied: The first modified the transformer architecture to be able to learn from ASTs and the second improved self-supervised pretraining for code-LMs with denoising tasks constructed from the code's syntax tree. The main result, the proposed SyntaxPT LM, achieved state-of-the-art performance in code understanding benchmarks at the time of its development. In the final chapter of Part I, a novel self-supervised training strategy for Contextualized Code Search (CCS) was introduced. By fine-tuning the SyntaxPT model on a large-scale codebase using contrastive learning, the SyntaxPT-ccs model outperformed traditional retrieval methods such as bm25. This fine-tuned code-encoder model demonstrated state-of-the-art performance not only in contextual code retrieval but also in other code understanding tasks from CodeXglue.

Overall, Part I has focused strongly on the design of machine learning models and their training task. Hence, evaluations of the research in Part I primarily aimed to measure the general code understanding capabilities of the models through standard machine learning benchmarks, as is common practice in machine learning research (Goodfellow et al. 2016). These evaluations are important to assess the performance in research settings and to optimize machine learning pipelines, hoping that they accurately reflect real usage scenarios. Nonetheless, they only provide a limited view of the model's practical utility. Part II of this dissertation will now shift the focus from research benchmarks to practical applications of the developed models. Here, the goal is to explore how SyntaxPT and

SyntaxPT-ccs perform in practical settings that reflect actual developer workflows and needs. We first look at the performance of SyntaxPT-ccs from an end-user's perspective. Two user studies will assess how effectively the model supports developers in practical coding tasks by finding relevant code snippets in a codebase. Next, in Chapter 7, we take a different perspective and determine if SyntaxPT can be used to evaluate the quality of variable names, which are an important factor for code readability and maintenance. To this end, we will assess whether the predictions of the self-supervised model reflect widely accepted best practices in software engineering, in form of established software engineering guidelines for naming variables.

6.1 Introduction and Motivation

The development of software is a complex and time-consuming process, and the ability to reuse existing code can significantly reduce development time and costs. Thus, developers are constantly exchanging knowledge, via mailing lists, forums, and open-source repositories, to find solutions to their problems. In this context, CCS has the potential to enable much more effortless access to these solutions, as it seamlessly integrates into the developer's workflow. While a developer writes code in their editor, CCS can analyze the context together with the cursor position and search a codebase (e.g., a company's software-repositories) for relevant code snippets that could be inserted at the current cursor position. Compared to natural language code search, this eliminates the need for the developer to manually specify a query and additionally utilizes the currently written code as an additional source of information. This consequently reduces the amount of effort required to discover relevant code. It is important to note that the code snippets found relevant by SyntaxPT-ccs are not synthesized but rather retrieved from existing, verified solutions written by other developers. In this way CCS can enable safe and secure code reuse (as opposed to code generation tools, such as ChatGPT or GitHub Copilot that often ignore these and for which the source of the generations remains unclear).

> The benefits of code reuse are familiar to experienced developers—compared with newly added code, reused code has already been tested & proven stable in production. Often, reused code has been touched by multiple developers, so is more likely to include documentation. This accelerates the interpretation of the module by developers who are new to it. (GitClear et al. 2024)

Having an easily approachable way to discover code is particularly more relevant in companies with multiple parallel software projects, where developers (or development teams) often work in isolation from each other, while working on similar problems. In such contexts, knowledge often remains within the teams, and valuable solutions from one project may never reach other teams that could benefit. CCS could bridge these gaps and share solutions across development teams.

6.1.1 Contributions

The self-supervised approach to CCS presented in Chapter 5 uses contrastive learning on a large-scale codebase without manually labeled data. Pairs for training the model are sampled from complete code files in a self-supervised manner and deleaking steps have been introduced to minimize overfitting. Until now, CCS models have been evaluated only using research datasets (Mukherjee et al. 2020; Dahal et al. 2022; **Villmow** et al. 2022), and the models' practical utility for software developers remains underexplored. Hence, this chapter aims to investigate whether the model and training pipeline proposed in Chapter 5 work in real development scenarios. This poses additional research challenges related with applying the proposed SyntaxPT-ccs model in practical software development:

- The model must be integrated into a user-friendly application that allows developers to interact with the model. This includes a User Interface (UI) concept, as well as a filtering concept to restrict the search results to specific directories, files, or programming languages. Preliminary user tests have indicated that developers may appreciate additional filtering options to restrict the search space.

- The model must be robust to variations in the user's query formulation. In contrast to benchmarks, users may formulate their queries differently, and the model must be tolerant to these changes. For example, the users may misplace the cursor or query the model with an incomplete line. The model has not seen such situations during training, but should still be able to provide useful results.

- The codebase must be indexed, which includes selecting potentially relevant code snippets from the files in the codebase to build the index. These snippets form the basis for the search results. Using every possible code snippet in the codebase is not feasible due to the large number of possible snippets (e.g., all possible subsequences of a file).

- Redundant search results, i.e., cases where the model suggest Lines 1–5 and Lines 2–6 from the same file, must be filtered out to ensure that the search results are concise and relevant. The attention span of a user is limited, and redundant results may distract from the relevant ones.

With these challenges in mind, this chapter aims to investigate the following research objectives:

- CCS follows a specific workflow: (1) write code context, (2) analyze the result list, and (3) adapt the solution into their own code. A key question is: How do developers assess the effort and difficulty of these individual steps?

- What particular use cases do developers see for the CCS in practice, such as exploring a new software project, reusing code they have written previously, and others?

Even though the self-supervised approach to CCS proposed in Chapter 5 reduces the time for query formulation to a minimum (since in fact no explicit query is required), CCS still requires the developer to shift focus from its editor window to comprehend and adapt the search results. Hence, it remains unclear, whether developers find the concept of CCS useful. To address this research gap, this chapter introduces a prototype application for CCS, called CodeBuddy. A screenshot of this application is shown in Figure 6.1. Using CodeBuddy, a practical evaluation of SyntaxPT-ccs through two user studies is conducted. In Study A, the impact of CodeBuddy on the efficiency of developers on student's programming exercises is assessed, and in Study B potential use cases in a corporate scenario are explored.

According to Hevner et al. (2004) this research aligns with the principles of design science, which is why this chapter closely follows the proposed methodology and guidelines. The authors define design science as the creation of purposeful, innovative, and novel artifacts for specific problem domains. Such an artifact is the proposed CodeBuddy prototype, which is, to the best of the author's knowledge, the first prototype that allows developers to interactively search with CCS. Hevner et al. (2004) state that "design is inherently an iterative and incremental activity" (p. 85), and "the design process [...] a Generate/Test Cycle" (p. 88). Such a cycle has been applied in the development phase of CodeBuddy, in which several iterative steps of implementation and testing with an experienced developer as an alpha user have lead to a sequence of user interface improvements and model refinements[1]. In this chapter, the design decisions and outcomes of these iterations are described. For example, the model refinements have led to additions to the self-supervised pretraining routine of Chapter 5, that aim to improve the robustness of the model when used by end-users. This chapter thus contributes to both the academic understanding and practical application of CCS models.

Hevner et al. (2004) emphasize that to demonstrate the utility of CodeBuddy for developers, a "thorough evaluation of the artifact is crucial" (p. 82), using "methodologies available in the knowledge base" (p. 86). In this chapter two of such evaluation methods are used: (1) a controlled experiment that measured both quantitative and qualitative performance improvements by using CodeBuddy among computer science students in Study A, and (2) an observational case study with a professional software development team to discover the practical impact of CodeBuddy in Study B.

In summary the key contributions of this chapter are:

[1] The iterative development of CodeBuddy was conducted with the help of an experienced developer from the AOE GmbH in Wiesbaden, Germany.

1. Present enhancements to the self-supervised pretraining approach from Chapter 5 that improve SyntaxPT-ccs's usability and robustness, when used in a zero-shot application with end-users.
2. Present an indexing strategy to discover and index candidate code snippets from the codebase and introduce a post-processing strategy to filter redundant results.
3. Introduce and open-source a software application to search and visualize relevant code passages in a codebase, called CodeBuddy (the first tool for interactive CCS).
4. Evaluate CodeBuddy in two practical user studies: (1) Study A is a controlled experiment that measured performance improvements when using CodeBuddy in fourth-semester computer science students working on simple algorithmic programming exercises, and (2) Study B is a case study in which a professional software development team used CodeBuddy during their regular work activities.

6.2 Related Work

Code search is a fundamental activity in software development. It enables developers to locate relevant code snippets, understand unfamiliar codebases, and reuse existing solutions. For a detailed description about related work on code search, please refer to Section 5.2. This section outlines the related work on user studies in the context of the usability and efficiency implications of AI tools in practical software development and code search. The first has recently been studied: GitHub Copilot has shown significant productivity benefits in practical settings (GitHub 2024). Studies report a 56% increase in task completion speed (Peng et al. 2023) and high acceptance of code suggestions (Dohmke et al. 2023; Ziegler et al. 2024). To circumvent copyright issues, GitHub experiments with a code reference feature that uses code search to link generated snippets to public repositories (Salva 2023). CCS, on the other hand, directly retrieves existing code snippets from a trusted source without generating new code.

Several empirical studies have explored how developers search for code and the tools they use, for which Grazia and Pradel (2023) provide a comprehensive survey. They report that these studies typically use the following methodologies: They may (1) observe developers behavior (Ko et al. 2006), (2) analyze log files (Bajracharya and Lopes 2012), or (3) conduct a questionnaire-based survey (Singer et al. 1997), or a combination of these methods. This chapter follows these methodologies and analyzes logged requests and ratings, but also ask for feedback from the participants using a questionnaire. Additionally, we will conduct expert interviews to gain insights into the practical implications of CCS in a professional software development environment. Grazia and Pradel (2023) also report that, while some studies exist that analyze usage patterns in large-scale (code) search websites (Bajracharya

and Lopes 2012; Rahman et al. 2018), such as Koders or Google Search, many studies explore a practical scenario and index a codebase that consists of multiple software projects, such as one from a company or organization (Panchenko et al. 2011; Sadowski et al. 2015). This chapter will create a similar scenario and focus on medium-sized codebases consisting of tens of repositories, as in Study B, a software company's codebase.

Most of these studies are about regular code search (e.g., with short queries of natural language or parts of code), and none have been conducted specifically for CCS. Bajracharya and Lopes (2012) analyzed over ten million usage logs from the Koders code search engine, and found that most queries are short, with 79% of users providing only a single search term. This indicates that developers find formulating queries cumbersome and prefer to use short, simple queries. This supports our "queryless" CCS approach, where the model automatically infers the search intent from the context and cursor position.

Similarly, Ko et al. (2006) conducted an observational study of ten Java developers performing maintenance tasks. They discovered that developers spend a large amount of their time navigating and searching for relevant code and often use tools such as `grep` and `find` (Grazia and Pradel 2023). Studies by Rahman et al. (2018) and Sadowski et al. (2015) investigated how developers use general-purpose search engines like Google for code retrieval. Rahman et al. (2018) found that code-related searches often require more effort—such as longer time, more result clicks, and query modifications—compared to general web searches. Sim et al. (2011) evaluated different code search approaches and found that while general-purpose search engines are effective for finding small code snippets or reference examples, specialized code search engines perform better when searching for larger components or libraries. However, general web search engines, such as Google, "are the most popular choice for code search and will continue to be like that, in all likelihood, because they are lightweight, easy to use, and have sophisticated web interfaces" (Rahman et al. 2018). Based on these findings, the participants that were not allowed to use CodeBuddy in the controlled experiment in Study A were explicitly provided with a directory with the codebase in order to employ tools like `grep` or `find` and allowed to use general-purpose search engines to find helpful code snippets.

Recall that such user studies have not yet been conducted for CCS, where research has focused on improving retrieval or code generation quality on research datasets, rather than practical applications (such as in Chapter 5). We want to address this gap and test self-supervised CCS in two practical user studies with developers.

6.3 Approach

This section describes how the aforementioned challenges are addressed. It first describes the user interface and implementation of the CodeBuddy demo application, then presents enhancements made to the self-supervised pretraining approach from Chapter 5,

Figure 6.1: The web-frontend of the CodeBuddy demo application. On the top a code editor and the user has the cursor placed inside a class `Date` (at the yellow bar). Once the user clicks the blue search button (middle), the context and the cursor position is encoded with SyntaxPT-ccs, which is trained to retrieve code snippets that fit in the context at the cursor position. The search results are displayed below, with the option to expand each code snippet to see more context. Below each code snippet, the user can provide feedback on the relevance of the snippet.

,width=]figures/codebuddy/web-frontend.png

and finally describes the indexing strategy and the proposed filtering of redundant results within CodeBuddy.

6.3.1 Demo Application

SyntaxPT-ccs is evaluated in two user studies, which require a user-friendly interface for developers to interact with the model and a backend to index the codebase, retrieve code snippets, and store user feedback. Therefore, the CodeBuddy demo application has been developed to enable developers to perform searches, visualize results, and give feedback regarding the results. To be as seamlessly integrated into developers' workflows as possible, the application includes an editor and supports two user interfaces. The first is a browser-based interface, shown in Figure 6.1, used by students in the controlled experiment of Study A. The second is an IntelliJ plugin, shown in Figure 6.2, used by developers in Study B, which provides a more convenient interface that is integrated in their usual work environment. Since Study B is conducted in cooperation with a software company on their codebase, the application has been dockerized to be deployed on a cloud-based infrastructure within the company's network to ensure data privacy and security.

Recall, that SyntaxPT-ccs is trained to retrieve code snippets that could plausibly be inserted at the cursor position. When a user clicks the search button, the SyntaxPT-ccs retrieves code snippets from the codebase. In CodeBuddy, the cursor plays an important role, since it indicates the model for which position code snippets should be received. For instance, positioning the cursor *inside* a class—just as in Figure 6.1—signals to the model that instance attributes or methods should be retrieved. However, when the cursor is positioned *outside* the class (e.g., Line 1, Col. 1), one could expect the model to retrieve imports, enums, or even another complete class. Note that this is a new User Experience (UX)-concept, and the author is not aware that this feature has yet been used in any retrieval applications. Hence, there hypothetically is a learning curve for the users.

In CodeBuddy the user can set filters over the filepath and language of the code snippets (which defaults to snippets in the same language as the current file). However, filters over the filepath have been used only by the professional developers in Study B (see text fields below the search button in Figure 6.2). Preliminary developer feedback revealed that snippets were often challenging to understand without additional context before and after the code snippet. To address this issue, the web-frontend displays three lines of code above and below each snippet, as shown in Figure 6.1. The IntelliJ-plugin, on the other hand, displays the found snippets without additional context, as shown in Figure 6.2. In both

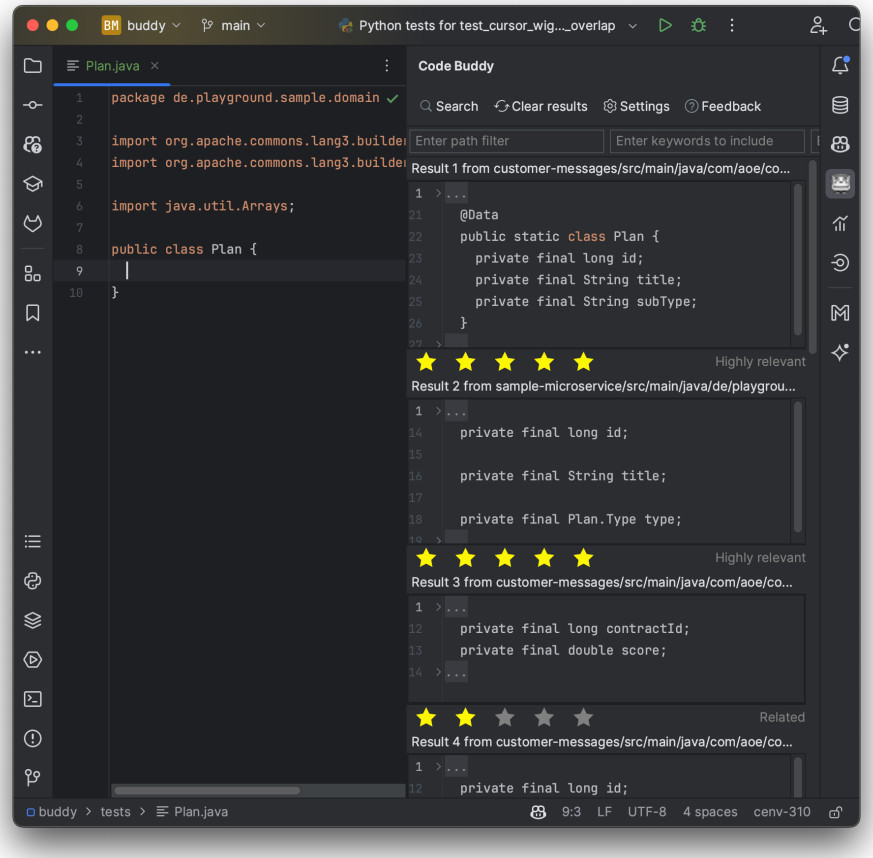


Figure 6.2: The IntelliJ-plugin frontend of the CodeBuddy demo application used in Study B. The regular IntelliJ editor is used to code and a side-panel with the CodeBuddy plugin can be opened to the right. Once the user hits search, the current cursor position in the editor marks the information need and results are retrieved. Note that at the top of the plugin, the developers can specify additional filters to restrict the results.

frontends the developer has an option to view more context if needed. Both frontends show the path to the file above each search result, and in the web-frontend, users can click on the file path to open the full file in a new tab. In both frontends, the relevant code snippets are presented with syntax highlighting and line numbers for best possible readability. Feedback can be given with a five-star rating system below each search result to enable users to rate the usefulness of the code snippets.

Software Stack

A high-level overview of the software system is shown in Figure 6.3. The web-frontend is implemented as a single-page application in Vue.js (You 2024), and the IntelliJ-plugin is implemented in Kotlin. Both frontends communicate with the backend via a REST API, which is implemented in Python using the FastAPI web framework (Ramírez 2023). The backend runs SyntaxPT-ccs and provides API endpoints for authentication, search

requests, and feedback. In addition to the REST API, the backend handles asynchronous tasks, such as indexing the codebase. Internally, the code files are stored in a MongoDB database, which also stores user management data and feedback. The code snippets are encoded with the model and the embeddings are stored in the Qdrant vector database (Zayarni 2023), which is used for efficient vector search operations. Along with each embedding, metadata to the snippet is stored in Qdrant, including the file path, line range, and language.

Before the application starts, it first indexes relevant code snippets from the codebase (this can be time-consuming, but has to be done only once). Therefore, CodeBuddy stores the code files in a MongoDB database, detects candidate snippets (this step is described in detail in Section 6.3.3), encodes the candidates with the model's target encoder (see Section 2.2.2), and stores the target embeddings in a vector database. When a user clicks the search button the code in the editor is encoded with SyntaxPT-CCS into a query embedding (see Figure 5.3). After obtaining the query embedding, it is matched with all target embeddings stored in the vector database using a filtered nearest neighbor similarity search (Malkov and Yashunin 2020; Douze et al. 2024). The vector database returns the file path and line ranges of the nearest code snippets in the latent space. These are first used to apply the non-maximum suppression algorithm, described in Section 6.3.3, to filter out redundant results, and subsequently query the actual code from the MongoDB database.

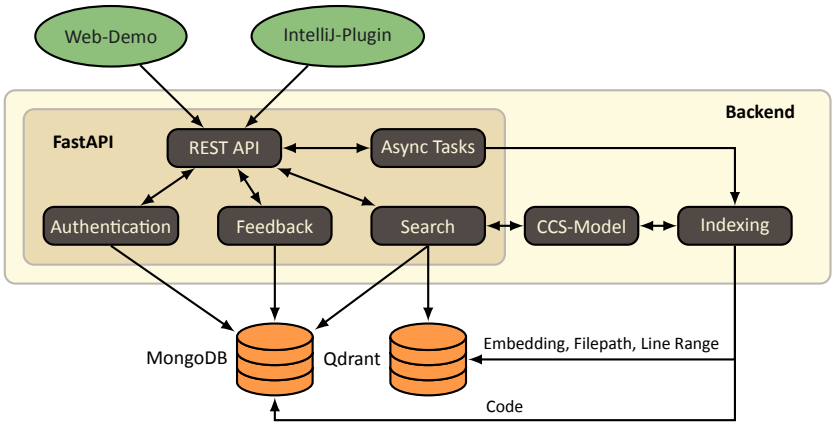


Figure 6.3: High-level overview over the architecture of the CodeBuddy demo application. The software stack is dockerized and can be deployed on any cloud-based infrastructure. The backend service is implemented in Python and uses the FastAPI web framework to provide a REST API for the frontend applications. The application runs the CCS model and handles asynchronous tasks such as indexing the codebase. During indexing the code is stored in a MongoDB database, while an embedding for each code snippet is stored along with metadata in a Qdrant vector database, which is used for efficient vector search operations. The frontend applications are a browser-based interface implemented in Vue.js and an IntelliJ-plugin written in Kotlin.

Feedback Mechanism

For feedback a five-point scale has been adopted from IR benchmarks. The scale used within CodeBuddy closely follows the descriptions in the four-point scale proposed by Craswell et al. (2020) for an IR benchmark for the Text REtrieval Conference (TREC) but has been worded to better fit the task. Additionally, a *remotely relevant* middle category is added to have a more fine-grained rating:

1 Star: **Irrelevant**: The snippet has nothing to do with my search.

2 Stars: **Related**: The snippet seems related to my search but does not answer it.

3 Stars: **Remotely Relevant**: The snippet provides some information relevant to my search, which may be minimal.

4 Stars: **Relevant**: The snippet has some answer for my search, but may need substantial modification.

5 Stars: **Highly relevant**: The snippet is dedicated to my search and can be used with minimal modification.

6.3.2 Model Enhancements

SyntaxPT-ccs introduced in Chapter 5 has a siamese architecture with transformer LMs as encoders. The model is trained using contrastive learning on context-target pairs, which are sampled randomly from code files. To prevent the model from overfitting (since context and target come from the same code file), several measures were taken: (1) *Tree-based span selection* samples only complete subtrees of the AST to be cutout as a target, so that the selected span is a syntactically valid code snippet. This prevents the model from solving the task by merely detecting syntactical matches, such as an opened but not closed bracket in the query. (2) With *mutual identifier masking* identifiers that are shared between context and target are masked in either the context or the target. At application time, query and target predominantly not share the same identifier names; thus, this forces the model to learn the intent of the code snippet rather than just connecting identifier names. (3) *Dedenting* is used to probabilistically remove leading whitespace from the target snippet, to prevent the model from returning only targets with matching indentation level.

An example of a context-target pair produced by this process is shown in Figure 6.5. This self-supervised learning procedure can be trained on large codebases without the need for labeled data. However, the pretraining always cuts-out the "exact" missing piece of code, which is always a complete substructure (see (1) above). Preliminary user tests have shown that these assumptions are violated when developers formulate their queries freely:

1. Users often typed an incomplete line and then ran a search, as if using a code completion tool. However, the model is trained to find only complete syntactical

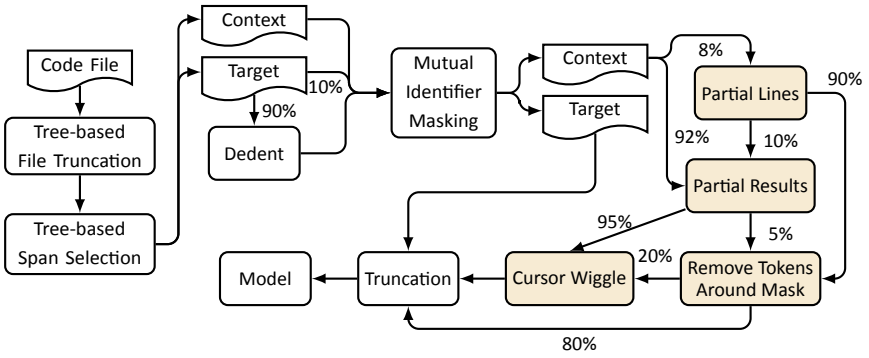


Figure 6.4: The self-supervised training pipeline for CCS with the additional steps to improve the user experience highlighted in yellow. Apart from these changes, this is the same pipeline as detailed in Section 5.3.2. In these new steps the context is modified to simulate user queries with incomplete lines and cursor variations, but also to allow finding partial results. If a step is only applied sometimes (e.g., to 8% of samples), its frequency is indicated on the arrow leading to the step.

units and not partial lines, because of the context-target pair creation with tree-based span selection. This leads to suboptimal results.

2. The model has not been trained to yield partial results that contain some—but not all—of the missing code. These results, however, may be interesting in practice. For example, if a user starts writing a method but most of its body, including the return statement, is missing, the model only finds results that complete the full method and thus contain a return statement. However, intermediate statements within the method could also be useful.

3. During pretraining, the exact token position where code is needed is known. In practice, however, users often misplaced the cursor at a slightly different position than where code was needed. We observed that the model strongly focuses on the cursor position, and even slight variations can result in significantly different search results. This behavior is suboptimal and can be difficult for users to understand.

Since the approach to CCS is completely self-supervised and no additional fine-tuning data is employed, these issues need to be addressed during pretraining. To this end, this chapter introduces several additions to the training pipeline of Chapter 5 that modify the context snippet to simulate queries with the aforementioned user behaviors. The new pipeline is shown in Figure 6.4, where the new steps are highlighted in yellow. These steps are applied probabilistically to some of the training samples, for which we estimated their frequency based on the observed user behavior during the preliminary tests. Note that we do not want to overrepresent these cases. Apart from the new steps, the model architecture, pretraining dataset, and hyperparameters remain the same.

```
1 def totalSalary(VAR1):
2     con = connect(host='localhost',
      ↪  user='root', database='db')
3     cursor = con.cursor()
4     |←CURSOR
5     return salary + bonus
6
7 print(totalSalary(1))
```

(a) Context

```
1 query = (
2     "SELECT wage, bonus"
3     "FROM employees"
4     "WHERE emp_no = %s")
5 VAR2.execute(query, (emp_no,))
6 row = VAR2.fetchone()
7 VAR3, VAR1 = row
```

(b) Target

Figure 6.5: Example of a context-target pair produced by the self-supervised learning strategy for CCS, as introduced in Chapter 5. The creation process of the pair has been visualized in Figures 5.3b and 5.3c.

```
1 def totalSalary(VAR1):
2     con = connect(host='localhost',
      ↪  user='root', database='db')
3     cursor = con.cursor()
4     query =|
5     return salary + bonus
6
7 print(totalSalary(1))
```

(a) Simulate incomplete queries by copying tokens from the target (Figure 6.5b) to the context (highlighted in blue).

```
1 def totalSalary(VAR1):
2     con = connect(host='localhost',
      ↪  user='root', database='db')
3     cursor = con.cursor()|
4     return salary + bonus
5
6 print(totalSalary(1))
```

(c) Remove tokens around the mask token to reduce focus on cursor position.

```
1 def totalSalary(VAR1):
2     con = connect(host='localhost',
      ↪  user='root', database='db')
3     cursor = con.cursor()
4     |
5
6 print(totalSalary(1))
```

(b) Remove everything behind the mask token to allow for partial results. The removal is limited to the scope of the mask token using the syntax tree.

```
1 def totalSalary(VAR1):
2     con = connect(host='localhost',
      ↪  user='root', database='db')
3     cursor = con.cursor()
4
5     return |salary + bonus
6
7 print(totalSalary(1))
```

(d) Change cursor position to simulate slight variations and misplacements.

Figure 6.6: User experience improvements to the approach for self-supervised CCS. Note that only the context is changed (i.e., the query), while the target remains the same for all samples. The idea is that the model should learn to find the same target code snippet, regardless of the changes in the context due to variation in user's queries and cursor placement.

Partial Lines To simulate scenarios where the context contains incomplete lines as in code completion, 8% of the samples are modified by adding a few tokens from the beginning of the target to the context snippet. This allows the model to find partially complete code snippets in scenarios where a user has started writing a line but not finished it. Specifically, the amount of tokens n_{pl} is sampled from round$(n_{pl}) \sim \mathcal{N}(0, 4)$ and added to the context before the mask token. A visualization of applying this step to the context in Figure 6.5a is shown in Figure 6.6a.

Partial Results In 10% of the samples a scenario in which the user has just started writing code and wants to find a partial solution is simulated by removing everything behind the mask token from the context. This removal is limited to the scope of the mask token using the syntax tree. For example, if the mask token is placed at the position of a top-level statement in a method, all following top-level statements in the method are removed. Take a look at Figure 6.6b, in which this step removes the return statement from the context snippet (Line 5 in Figure 6.5a), but keeps the out-of-scope print statement (Line 7 in Figure 6.5a). This enables the model to learn that, even though the return statement is missing, the return statement does not have to be part of the retrieved target snippet.

Cursor Position Users often placed the cursor at a different position than where the code was needed. To create a more robust model that is less sensitive to slight variations in cursor position, two new steps are introduced, which are denoted as *Remove Tokens Around Cursor* and *Cursor Wiggle* in the pipeline in Figure 6.4. The first is visualized in Figure 6.6c and aims to reduce the model's focus on the exact syntactical position of the mask token in the context snippet. In 5% of the samples, a random number of tokens directly around the mask token are removed. This is achieved by removing a window of $\lceil n_{rt} \rceil \sim \mathcal{N}(0, 3)$ tokens around the mask token. The second step adds noise to the cursor position in 20% of the samples. This is achieved by shifting the mask token $\lceil n_{cw} \rceil \sim \mathcal{N}(0, 2.5)$ tokens to the left or right. This step is visualized in Figure 6.6d.

6.3.3 Indexing and Retrieval

During inference, the model searches for appropriate target code snippets in the codebase given the user's editor context. For instance, searches may target a method body, a statement, or multiple statements within a code file. However, at indexing time, this level of granularity is unknown, which theoretically requires indexing code snippets at *all* possible granularities. Clearly, this is not feasible. To address this problem, the AST is used to determine which parts of the files will be indexed as possible targets. Note that this is similar to the context-target pair creation during pretraining (see Section 5.3.1). Specifically, a subset of 24 node types of the set of nonterminal labels $\mathbb{L}$ is manually selected to be relevant targets (see Section 2.1.2), e.g., `ExpressionStatement`, `IfStatement`, etc. The

complete list is provided in Table B.1. This subset has been primarily constructed for Java code, but has been found to also work well for Python, TypeScript, and JavaScript.

Each selected node is a separate syntactical unit (e.g., Lines 1–4, and Line 5 in Figure 6.5b are two separate nodes). When these nodes are considered only individually, the user can never find top-level multi-statement code snippets (such as the complete Figure 6.5b that is composed of multiple expression statements). Hence, adjacent sibling nodes in the AST (e.g., two lines of code) are combined up to a maximum of six nodes and are also indexed. This approach is analogous to word-level n-grams but for tree-nodes. Pretraining uses a similar strategy during tree-based span selection, as described in Section 5.3.1.

In an offline step, all those code snippets are encoded using the target encoder of the model and the embeddings are stored in the vector database. Note that opposed to pretraining, no deleaking steps or modifications are applied to the selected code snippets at inference time (indexing or retrieval). At runtime, the user's editor context with the cursor position inserted as a mask token is encoded by the context encoder, resulting in the query embedding. This query embedding is compared with all indexed target code snippet embeddings in the database using nearest neighbor searches (see Section 2.2.2)[2].

Note that this strategy encodes every possible target code snippet individually and independent without the remaining context of the file, even though the context could provide valuable information during semantic matching. Obviously, this hardens the retrieval task and is more error-prone. Future work could explore combinations of SyntaxPT-ccs that additionally use the contextual information in the target file. For example, first find matching the contexts with some model, then locate the relevant parts within the matching contexts with SyntaxPT-ccs.

Non-Maximum Suppression

Preliminary user tests indicated that the model frequently returns overlapping code snippets, such as lines 4–8, 6–8, and 5–10 from the same file. This is reasonable, since the overlap in content of these snippets can be expected to result in similar embeddings. Presenting redundant results to the user is expected to negatively affect the overall user experience. This is not only because "most users [...] rarely go to the second page of [10] results, and most of the time they only click on one document in the result set" (Pan et al. 2007, p. 803), but also the cognitive load associated with reading and evaluating multiple redundant solutions can be mentally straining. Consequently, it is essential to minimize overlapping code snippets in the result list.

[2] In practice the vector database employs an approximate graph-based algorithm for fast nearest neighbor searches in high-dimensional spaces, such as HNSW (Malkov and Yashunin 2020). Within the algorithm it is possible to trade-off accuracy for speed or memory. In the demo application the setting with the highest retrieval accuracy is used, regardless of speed and memory.

To this end, this chapter proposes to post-process SyntaxPT-ccs's search results using *Non-Maximum Suppression (NMS)* (Canny 1986) to filter out redundant code snippets. This method is commonly used in visual object detection (Viola and Jones 2004, p.159), where initially a large set of bounding boxes is predicted, followed by a post-processing step to filter or merge redundant predictions, retaining only the highest-scored ones (Hosang et al. 2017). Xia et al. (2019) utilized NMS for the NLP-task named entity recognition, but to the best of the author's knowledge, this is the first time it has been applied in CCS or code-retrieval applications.

The non-maximum suppression algorithm filters the search results in the following way: First, snippets are sorted in descending order according to their relevance scores. The next snippet with the highest score, X, is then iteratively selected, and all other snippets Y in the result list that originate from the same file are discarded if they overlap of more than 50% with X. Thereby, overlap is measured as the Intersection over Union (IoU), which is computed based on their line ranges. Formally, if $[s_1, e_1]$ and $[s_2, e_2]$ represent the start and end line ranges (inclusive) of two code snippets X and Y, their intersection I is calculated as:

$$I(X, Y) = \max(0, \min(e_1, e_2) - \max(s_1, s_2) + 1) \tag{6.1}$$

The IoU is subsequently defined as:

$$\text{IoU}(X, Y) = \frac{I}{(e_1 - s_1 + 1) + (e_2 - s_2 + 1) - I} \tag{6.2}$$

After non-maximum suppression, the top 10 most relevant code snippets are presented to the user as a result list, ranked by descending scores.

6.4 Study A: Programming Exercises

Research Question 6.1: *Are programming students more efficient when using CodeBuddy and a codebase where solutions to similar problems exists, compared to students who do not use CodeBuddy but have access to the same codebase and can use conventional search methods, including web search?*

The primary goal of Study A is to evaluate the impact of CodeBuddy on the efficiency of programmers in a controlled environment with a considerable number of participants ($n = 41$). The study aims to explore whether students using CodeBuddy display objective and subjective improvements in task efficiency compared to those using conventional search methods, such as web search engines or a local file search. The main hypothesis of the study is that students using CodeBuddy will complete programming tasks more

efficiently than those who do not use CodeBuddy, as measured through both subjective self-assessment and objective scores rated by a teaching assistant.

To test this hypothesis, the students work on modified algorithmic programming exercises from the lecture "Algorithms and Data Structures". The codebase consists of student submissions from previous years on similar and related exercises and some additional algorithmic software projects. Efficiency is measured by the students' ability to complete programming tasks within a fixed time frame. The students are divided into two groups: one group uses CodeBuddy exclusively, and the other can use traditional search methods. The students can validate the correctness of their implementation through automated tests in a submission system. Also, students were asked for their subjective impression to to what degree they solved the respective exercises, and a teaching assistant familiar with the exercises provided an objective grading on the final submission. For CodeBuddy to improve student efficiency, it must (1) effectively retrieve relevant code snippets, and (2) students must comprehend and interpret the search results, to (3) successfully integrate the solutions into their own code.

6.4.1 Experimental Setup

The study was conducted in three rounds with separate groups of students of in total 41 participants. The students were in their fourth semester and had taken the lecture "Algorithms and Data Structures" a year ago (2nd semester, 5 credit points). At the beginning of each session the students were given a short introductory presentation to CodeBuddy with a short example of how to use the tool and could try it out. The student were also advised about the aims, overall purpose, and methods of this study. They were guaranteed anonymity of their data produced in the experiments. The students worked alone, no pair programming, and implemented their solutions in a browser-based code editor within an online submission system[3], which is routinely used at HSRM, and the students were familiar with. They could test their solution with the system as often as they needed which verifies it against automated test cases. Each student worked on three exercises for a maximum of 15 minutes per exercise. When the processing time was over, the last state of their work was submitted to the submission system. After completing each exercise, students were shown the correct solution of the exercise to self-assess their performance. At the end of the study, students filled out a questionnaire, and their submitted solutions were graded by a teaching assistant who was unaware of the group assignments.

Exercises Each student worked on three exercises, which were slight variations of old exercises from the lecture. Thus, the exercises were not completely new to the students. For example the students had to implement logic for a syntax parser that validates brackets

[3]The submission system can be accessed at https://subato.cs.hs-rm.de.

The following method determines all primes smaller than n:

PROCEDURE PRIMEFILTER(n)
1. Write down all numbers from 2 to n in sequence.
2. Mark the number 2 as prime and strike out all multiples of 2.
3. Find the smallest number m that is neither struck out nor marked.
4. Mark m as prime and strike out all multiples of m.
5. Repeat steps 3–4 until no number $m \leq n$ can be found.
6. All marked numbers are prime numbers.

Implement the following method(s):
- `boolean[] filter(int n)`: This method should return a boolean array `b` of length $n + 1$, which indicates for each number $k = 0, \ldots, n$ whether it is a prime number or not. (The index in the array serves as a reference to the numbers)

(a) Task description (translated to English).

```java
public class Primes {
  static boolean[] filter(int n) {
    boolean[] array = new boolean[n + 1];
    for (int i = 0; i < array.length;
    ↪ i++) array[i] = true;
    for (int j = 2; j < array.length;
    ↪ j++) {
      // FIXME
    }
    return array;
  }
}
```

.java

(b) Code skeleton provided to the students.

Figure 6.7: The prime number filter exercise given to the students, in which the students need to implement logic for the Sieve of Eratosthenes.

in a string, a sorting algorithm such as Mergesort or Heapsort, or a prime number filter using the Sieve of Eratosthenes algorithm. The prime number filter task is visualized in Figure 6.7 and the complete list of exercises shown in Table 6.1. The exercises were created by another teaching assistant unaware of the performance of CodeBuddy. The assistant was advised to strip down the exercises to a manageable size for the 15 minute time-frame, and change variable, method, and class names, so that keyword matching with previous solutions would not be sufficient to solve this task. The students were provided with skeleton code files, such as the one in Figure 6.7b, in which the logic for filtering had to be implemented. For instance, the original class and method in the prime number example were called `Sieve` and `sieve`, respectively, which were renamed in the skeleton in Figure 6.7b to `Primes` and `filter`, respectively. Note that the students had encountered the original versions of these tasks a year earlier. This situation resembles that of developers who look up solutions for similar—but not identical—problems.

Search Index The search index for this experiment was designed to contain realistic solutions to the given programming exercises and should also be large enough to pose a challenging retrieval scenario. Since the exercises were derived from old exercises of the same lecture, old student submissions of the original exercise could be used as relevant solutions for the derived exercise. Recall that because identifiers were changed in the skeleton, a simple keyword matching was not an effective approach to find the old solutions. From the old submissions five more or less diverse solutions were manually selected to be part of the codebase. Two of such solutions for the task of Figure 6.7 are shown in Figures B.1 and B.2. To obtain a realistically-sized codebase, additional code files were added to the index: A Java Algorithms package that implements various algorithms and

Task	Description
Syntax Parser	Implement a method that validates brackets using a stack.
Binary Tree	Implement a method to verify if a binary tree is complete based on its depth.
Quicksort with Stacks	Implement Quicksort using stacks, so that the smallest element is at the top of the stack after sorting.
Primes Filter	Implement the Sieve of Eratosthenes to find all prime numbers below a given limit.
Quaterly Search	Implement a variant of a binary search that divides the array into four parts.
Ternary Search	Implement a variant of a binary search that divides the array into three parts.
Heap Sort	Implement HeapSort for integers using a Max-Heap for ascending sorting.
Logic Solver	Implement methods that validate logical formulas in conjunctive normal form.
Double Linked List	Implement some methods for double linked lists.
BigInteger Search Tree	Implement recursive insertion and counting of smaller elements in a Search Tree.
Mergesort	Implement the `merge()` method for an ascending Mergesort in Java.
Double Linked Queue	Implement descending priority queues with double linked lists.

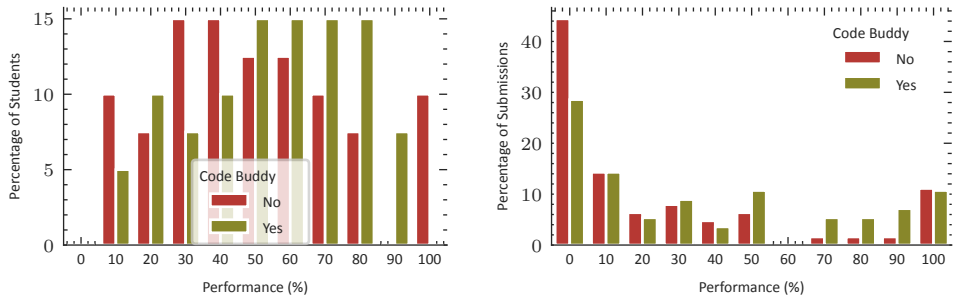
Table 6.1: List of programming exercises used in Study A.

data structures, as well as five random solutions from every other exercise of that lecture that were not part of the chosen exercises. This resulted in a total of 1962 files in the search index, from which 625,128 snippets were extracted and indexed.

Control Group At the beginning of the session, the students were randomly divided into two groups. The first group worked on the exercises without CodeBuddy. To ensure a fair comparison this group was provided with the files in the codebase locally on their computer (e.g., in order to use `grep`). This group could also perform traditional web-based searches, e.g., to obtain solutions from Google or Stack Overflow. The other group used CodeBuddy exclusively to complete the exercise (i.e., no local searches or web searches were allowed), the students could query the search index as often as they wanted and were asked to provide feedback for each search result they found useful. For both groups, using code generation tools such as GitHub Copilot or ChatGPT was prohibited. After two exercises, the groups were switched, and the students that previously worked with CodeBuddy now work without it. Note that both groups worked on the same exercises in the same order. In summary, one half of the students worked on two exercises with CodeBuddy and one without, while it was vice versa for the other half.

6.4.2 Results

With RQ 6.1 this study aimed to assess efficiency improvements of students using CodeBuddy compared to those who did not. To answer this research question, the objective performance and the subjective, self-assessed performance of the students is analyzed, grouped by whether the students used CodeBuddy or not. To correctly estimate their own performance in the self-assessed rating, the students were shown the correct solution after each exercise and at the end asked to estimate their overall progress with and without CodeBuddy. Their estimate is visualized in Figure 6.8a. For the objective performance



(a) Subjective ratings of performance by students.

(b) Performance based on scores given by the teacher.

Figure 6.8: Results of Study A comparing the performance of the students with and without CodeBuddy. In (a) it can be seen that the students using CodeBuddy rated their performance slightly higher than those who did not, and in (b) also received higher scores from the teacher (who was unaware whether the students used CodeBuddy or not).

a teacher graded the performance of each student in its final submission in 10% steps, which is shown in Figure 6.8b. Recall that each student worked on three exercises, which is why this is visualized per submission. Apart from performance metrics, the students provided feedback about their experience with CodeBuddy and CCS in general in a final questionnaire. Following common practice, the students rated their agreement with the statements in the questionnaire about their experience with CodeBuddy on a commonly used five-level Likert-scale (Likert 1932): (1) strongly disagree, (2) disagree, (3) neutral, (4) agree, (5) strongly agree.

For validating the significance of the results, the non-parametric Mann-Whitney-U-Test (Mann and Whitney 1947; Wilcoxon 1945) is used. The Mann-Whitney-U-Test is a test between two groups on a single ordinal variable, i.e., the students working with and without CodeBuddy, measured by their performance rating. This test is chosen over the common t-test, since the t-test requires a normal distribution of the variable, whereas the Mann-Whitney-U-Test does not assume any specific distribution. The null-hypothesis $\mathcal{H}_0$ is that there is no significant difference between the medians of the two groups, which can be rejected with a 5% significance level, when the Mann-Whitney-U-Test returns a p-value smaller than 0.05. Then the alternative hypothesis $\mathcal{H}_A$ can be accepted, i.e., that there is a significant difference between the two groups.

From Figure 6.8a it can be seen, that the students who used CodeBuddy rated their own performance slightly higher (mean = 54.5% $\pm$ 23.1) than those who did not use CodeBuddy (mean = 49.8% $\pm$ 25.9). However, with a p-value of 0.293 the difference in the distributions of the two groups of n_1 students working with and n_2 students without CodeBuddy is not significant (Mann-Whitney U=909, n_1=n_2=41, p-value > 0.05, two-tailed). Median subjective performance ratings of the two groups were 60% and 50%, respectively. The similar subjective performance may be due to the fact that the students

had to familiarize themselves with the new tool and development environment, which could have potentially also worsened their performance. Additionally, CodeBuddy provides different types of information. Instead of using a familiar web search to obtain information (that may return sites explaining the algorithm, and offer potential solutions), the students had to rely fully on CodeBuddy (that only returns partial solutions, without explanations). Hence, from a subjective perspective CodeBuddy seems to provide the same amount of useful information compared to what traditional web searches offer. One would have to conduct another study with a third control group, that is not allowed to search for any information, to measure the absolute impact of the retrieved information.

A different effect could be observed for the objective performance as rated by the teaching assistant in Figure 6.8b. Recall that the assistant rated the performance blindly without knowing whether the solution was implemented with or without CodeBuddy. Here, one can see that the performance on exercises had been significantly higher when the student had been using CodeBuddy (mean = 37.1% ± 36.6) compared to when not (mean = 25.1% ± 34.0). A p-value of 0.0498 indicates that the distributions of the two groups of n_3 exercises completed with CodeBuddy and n_4 exercises without CodeBuddy differed significantly (Mann-Whitney U=2122, n_3=56, n_4=63, p-value < 0.05, two-tailed). The median objective performance of the two groups had been 30% and 10%, respectively. These findings revealed that the students self-assessed their performance higher than what could be objectively measured. The rather low overall performance of the students could be attributed to the short 15-minute timeframe per exercise.

Concluding RQ 6.1, the controlled experiment indicates that—according to objective performance measuring—students indeed are more effective when using CodeBuddy instead of traditional search methods. This is a promising finding, since it means that in order to have a higher performance the students needed to find relevant solutions with CodeBuddy and be able to comprehend these solutions. However, subjectively the students rated their performance similarly. Additionally, the model did not work perfectly on all occasions. On one exercise that aimed to implement a recursive binary search by splitting an array into four parts, CodeBuddy retrieved many solutions to a similar binary search implementation splitting into three parts. However, while this was close to what was needed, no actual solution to the original problem was retrieved. Even though the solution was similar to what they needed, the students took longer to adapt the solution and were unsatisfied with the results. But overall for most tasks at least one relevant solution was retrieved.

Apart from performance metrics, the students were also asked to provide feedback about their experience with CodeBuddy and CCS in general. Figure 6.9 visualizes the questions and results of the questionnaire. Most students found relevant results in CodeBuddy (mean = 3.4 ± 1.1), which is also reflected in the feedback to the search results where the

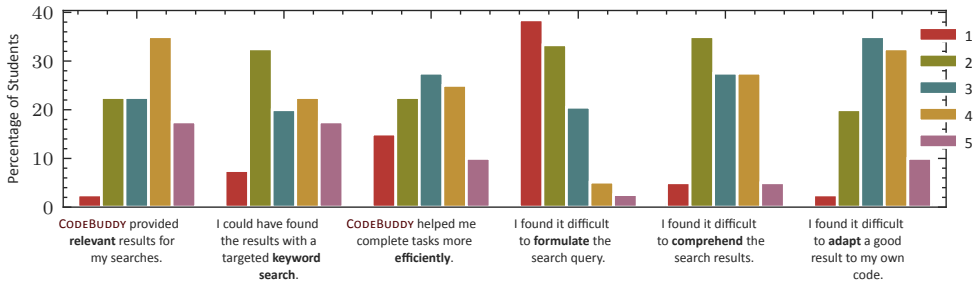


Figure 6.9: Results of the questionnaire in Study A. The students rated their agreement with the statements on a Likert scale from 1 (strongly disagree) to 5 (strongly agree).

MRR of the results which the students rated as most useful was 52.4%[4]. Students found that it helped them complete tasks more efficiently (mean = 2.9 ± 1.2). Students reported that formulating queries in CodeBuddy was relatively easy (mean = 2.0 ± 1.03). However, they found it more challenging to judge the relevance of the search results (mean = 2.9 ± 1.02). Especially the adaption of the solution to the students' current coding context had been perceived as stressful and time-consuming (mean = 3.3 ± 0.99). This is understandable, as at the time of the experiment many students have become used to the tailored solutions of code generation LMs. This leaves potential for future work that can automatically adapt the relevant solutions of CodeBuddy to the current context with LMs, for example by using RAG. When code generation tools are guided with external knowledge, research shows that hallucinations can be reduced (Huang et al. 2023).

6.5 Study B: Corporate Scenario

Research Question 6.2: *What use cases do professional developers see for CodeBuddy in their daily work?*

The goal of this research question is to evaluate the potential usefulness of CodeBuddy in a corporate development environment. This setting is particularly interesting from a research perspective, as the self-supervised CCS model has been trained only on open-source code. Corporate codebases are often highly domain-specific, and if CodeBuddy is able to retrieve relevant code snippets in this context without additional fine-tuning, it would demonstrate the model is applicable to a wide range of possible scenarios.

To address RQ 6.2, a case study was conducted with four developers from a professional software development team that integrated CodeBuddy into their everyday work. The company has a typical agile development environment where multiple semi-independent teams work on the same or related products. These teams frequently develop similar

[4]When computing metrics such as MRR with the five-star feedback scale introduced in Section 6.3.1, any rating of three stars or higher is considered relevant (Craswell et al. 2020).

features or face similar challenges. CodeBuddy was deployed within the company's infrastructure, and a snapshot of the company's codebase was indexed. This snapshot comprised approximately 7,000 TypeScript files, 4,500 Java files, 2,500 JavaScript files, 700 PHP files, 10 Rust files, and 5 Go files. Among the four participating developers, two were frontend developers, that worked primarily with Java code, and two were backend developers, that wrote JavaScript and TypeScript code. One member from each frontend and backend team was experienced, while the other one was new to the team and relatively unfamiliar with the codebase. The developers were instructed to use CodeBuddy as frequently as possible, particularly when they encountered an information need (which—they believed—could be satisfied with code from the codebase). When using CodeBuddy, they were asked to rate as many search results as they could. The study spanned over three months, from mid-January to mid-April 2024. At the end of the study, individual expert interviews were conducted with each developer, and their search queries and feedback were analyzed.

6.5.1 Results

RQ 6.2 aims to detect potential use cases for CodeBuddy in a corporate environment. Hence, first it should be analyzed whether CodeBuddy provides relevant results in domain specific environments, since, as previously detailed, CodeBuddy had been pretrained only on open-source code and is used now in a zero-shot fashion on domain code. An MRR of 41.2% for the most useful search result confirms that CodeBuddy also works in corporate application scenarios, while it leaves room for improvement. Interestingly the interviews revealed, that the frontend developers were more satisfied with the results than the backend developers. This may be because of the similar technological stack used throughout the different frontend products of the company, while the backend technology was rather specialized. In specialized domains, the codebase is more likely to not contain relevant code snippets, which is a limitation of the CCS approach. One frontend developer noted that when relevant code snippets were present, the results were often very close to what they needed, though not always exact matches. One of the backend developers mentioned that the quality of results could be hit or miss and some searches returned only irrelevant snippets. This is in line with the observation in Study A, where for one exercise the model retrieved exclusively solutions to a different exercise, while it worked well on most others. Future work could filter the search results to exclude results with low relevancy and rather show no results at all. Currently, the application always returns the top-10 results, even if their scores are low. However, when search results were not directly relevant or when results depended heavily on accurate cursor positioning, the developers quickly became frustrated. When analyzing the searches, one could see that 62.1% of the searches with feedback had at least one good result (22.2% of all searches).

In the interviews, the developers were directly asked what potential use cases they see for CodeBuddy. In addition to CodeBuddy's main use case, quickly finding relevant snippets within the codebase without extensive manual search, developers saw potential in using CodeBuddy to maintain consistency across projects, because with CodeBuddy they stumbled over similar and helpful code written by other teams. One of the newer developers used CodeBuddy in an exploratory way to familiarize themselves with the codebase. The developer found that CodeBuddy saved them time, because otherwise he would have been browsing the repositories manually. The ability to additionally formulate their intent in a natural language query had been requested by two developers. Some developers wished to search not only for code snippets that could fit in the current context but also for code snippets that are similar to the current context (similar to clone detection). The direct adaptation of a relevant code snippet to their current coding context was seen as a valuable feature. Hence, a combination with code generation tools could be a promising next step.

Furthermore, in modern software development—particularly in frontend projects—a considerable amount of work is done in configuration files. These are often scattered throughout the codebase, with and can often be found under many filenames, which makes manual search complicated. Additionally, the developers mentioned that code generation tools such as ChatGPT often struggle with configuration files. They wished CodeBuddy supported such files, so that they could easily search within other teams' configurations. They stated that finding in-use configuration files, which are validated and thus reliable, would be extremely helpful. Unfortunately, searching for configuration files is currently not possible with CodeBuddy, as the SyntaxPT model was pretrained exclusively on files in the 16 programming languages listed in Table 4.1. Adapting it to configuration files should be relatively straightforward. Configuration files are typically written in JSON, YAML, or other tree-like languages, which can be parsed and tokenized with the method introduced in Section 4.5.4. To adapt the CCS approach from Chapter 5 to configuration files, the tree-based span selection and dedenting technique could directly be used on these languages. One could hypothesize that mutual identifier masking would not be necessary, since there are no variables in such descriptive languages.

One problem noticed by the developers was that full files sometimes yielded less relevant results, compared to a (manually shortened) smaller context. This can be potentially attributed to the model's input sizes. During pretraining, sequences are truncated to a maximum length of 512 tokens (as detailed in Section 5.3.2). At inference time, however, the full editor context regardless of its length is encoded by the query encoder, which is made possible by the bucketed relative positional embeddings of the T5 architecture, as previously detailed. However, this potentially creates a discrepancy between pretraining stage and inference time. One could think of another subsequent, possibly self-supervised training with longer sequence lengths to circumvent this problem.

In addition to the exploration of potential use cases in the interview, the questions from the questionnaire in Study A regarding the usability of CodeBuddy had been asked. One developer found it straightforward to formulate search queries and appreciated the intuitive interaction through the cursor in the editor. However, others found the concept of CCS more challenging and were confused about positioning the cursor correctly for optimal results. The professional developers found it manageable and not tedious to read, understand, and evaluate the search results, especially when developers were familiar with the codebase. However, they found that adapting the solutions to their context as too effort-intensive, and one developer noted that most results served as inspiration rather than direct solutions. Specifically, difficulty in formulating the query achieved a mean of 2.5 ± 0.87 (std), while understanding the search results was rated by a mean of 2.0 ± 0.79, and adapting the results to their own code achieved a mean of 3.0 ± 1.5.

6.6 Conclusion and Future Work

This chapter explored the potential of CCS in real-world developer environments. It introduced CodeBuddy, a CCS demo application, and outlined how the self-supervised training of Chapter 5 could be improved for better handling of real-world developer behavior. Two studies validated the effectiveness of CodeBuddy: a controlled study with computer science students and a field study with professional developers. In Study A significant improvements in efficiency of the students when using CodeBuddy could be observed. In Study B, professional developers found CodeBuddy useful to facilitate code reuse across development teams by easily discovering relevant code. However, they also faced challenges with CodeBuddy's dependency on accurate cursor positioning and the relevance of search results.

This chapter has evaluated the practical utility of SyntaxPT-ccs through two user studies, and demonstrated the potential of CCS in real-world software development scenarios. To this end, it introduced CodeBuddy, a CCS demo application, and outlined how the self-supervised training of Chapter 5 could be improved for better handling of real-world developer behavior. The findings from Study A indicate that students using CodeBuddy were able to complete programming tasks more effectively, achieving significantly higher objective performance scores compared to those using traditional search methods. This suggests that CCS can enhance learning and efficiency in educational settings. In Study B, professional developers identified valuable use cases for CodeBuddy within a corporate environment. They found it particularly useful for exploring unfamiliar codebases, maintaining consistency across projects, and discovering existing solutions to common problems. The positive feedback demonstrates the model's applicability even when used in a zero-shot setting on domain-specific codebases.

However, the studies also highlighted areas for improvement. Users occasionally were confused about the cursor positioning, and found adapting the retrieved code snippets to their specific contexts cumbersome. This suggests that further research is needed to enhance the usability of CCS. Additionally, the model's performance varied depending on the domain specificity of the codebase (frontend/backend), so that further fine-tuning on the target codebase could improve relevance of the suggestions.

Integrating CodeBuddy with code generation LMs seems to be a reasonable next step, since this and the previous chapter showed that SyntaxPT-ccs is applicable not only in research contexts, but also in corporate environments. Generative LMs like GitHub Copilot and ChatGPT (GitHub 2024; OpenAI 2024a) have gained popularity in recent years, and are currently changing software development practices. Many developers now routinely use these tools for code suggestions. However, these models often fall short in more complex software projects, particularly because they lack specific knowledge about project internals or coding standards. This leads to suggestions that may not adhere to the corporate coding style or practices, and developers often spend additional time adapting the generated code to their needs. Furthermore, the source of the generated code is not always traceable, which raises concerns about the legal implications of using code produced by generative LMs (Salva 2023). In the context of code generation, RAG has become an increasingly popular way to improve the relevance of the code completions. In RAG additional contextual information, such as the content of other open files, the structure of directories, and more, is added to the prompt of the LM.

Two ways of integrating CodeBuddy with code generation tools could be considered. First, CodeBuddy could potentially be used as a retriever-backbone in a fully-automated RAG pipeline. It would enrich prompts for the LM with relevant code from a personal or the companies' codebase, to better adapt the generations to the coding style of the developer or the organization. This could potentially improve the quality of the generated code and reduce the need for manual adaptation. However, it does not allow the user to manually discover a specific solution, the search results are analyzed by the LM and the developer never interacts with the retriever itself. This could be explored in a second way, since the developers found CodeBuddy useful for exploring the codebase. After manually discovering a relevant solution with CodeBuddy, an instruction-based LM could then be prompted to adapt the snippet to the developer's context. Another option worth exploring is that CodeBuddy could directly present the adapted code snippets instead of raw search results to the developer. Both methods allow users to validate the code generation by viewing its source, verify license compliance, and reduce the need for manual adaptation.

Also, currently CodeBuddy encodes the target code snippets in isolation from their context (see Section 6.3.3), which makes the retrieval of relevant code snippets much

harder. Obviously, the context of the code snippet can provide valuable information about the code snippet itself. However, due to the nature of the self-supervised training, the context of the target snippet is not available during training. One could think of ways to make the target encoder context-sensitive, for example by using small amounts of supervised training data. Also, the non-maximum suppression could be replaced by a union of all results in the file, which could potentially improve the quality of the results. However, these are only ideas and would need to be evaluated in future work. To evaluate such strategies the Cocos benchmark introduced in the last chapter could potentially be used. While the last chapter focused on a fixed set of candidate snippets, the evaluation could be extended to select targets freely from the codebase. This would allow evaluating the quality of the retriever in a more realistic setting.

I've developed a new programming language! [...] Just normal code. Good clean syntax. Nothing weird. [...] Except the only variable name is "X". To refer to different variables you have to write "X" in different fonts.

— Munroe (2020)

7

Spotting Identifiers that Violate Naming Guidelines

Part I of this thesis introduced the SyntaxPT model, a transformer-based encoder-decoder code-LM trained on an AST-based multi-task learning objective. In the last chapter, a specialized version of this model was used to guide developers by introducing CodeBuddy, the first application for interactive CCS. In this chapter, we explore another practical application scenario of the SyntaxPT model's capabilities: its potential to improve the quality of source code by assisting developers in writing better identifiers. To this end, we will assess whether the SyntaxPT model has learned to follow widely accepted best practices in software engineering, specifically in relation to established software engineering guidelines for naming variables. The overall research goal of this chapter is to investigate whether LMs can be used to evaluate the quality of variable names, which are an important factor for code readability and maintenance.

7.1 Introduction

Metrics to measure the quality of source code range from negatively correlated cyclomatic complexity, Lines of Code (LOC), and nesting depth, to positively correlated number of

This chapter is adapted from **Johannes Villmow***, Viola Campos*, Jean Petry, Amine Abbad Andaloussi, Adrian Ulges, and Barbara Weber (2023b). How Well Can Masked Language Models Spot Identifiers That Violate Naming Guidelines? In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*. IEEE, pp. 131–142, previously published by ©2023 IEEE.

comments and the absence of linguistic antipatterns (Fakhoury et al. 2020). These aspects are important to understand the complexity and maintainability of source code. Another key aspect beyond such structural properties is the choice of identifiers: identifiers are the part of source code that developers can directly influence and interact with. They are typically written in natural language, are used to embed concepts, and have been found to be essential for communication between developers and code readability (Fakhoury et al. 2020). In software engineering, a large amount of time is spent understanding code written by others, particularly during maintenance, code reviews, debugging, and testing (Brooks 1983). Therefore, reduced comprehension time directly lowers development costs.

Bad or low-quality identifiers have been found to negatively impact the overall quality of a software project (Butler et al. 2010). This is supported by the findings of Fakhoury et al. (2020), who reported "evidence of a correlation between the quality of identifiers [...] and the quality of a software project". Arnaoudova et al. (2016) found that low-quality identifiers exhibit linguistic antipatterns, which "significantly increases the cognitive load" (Fakhoury et al. 2020) of developers. In contrast, high-quality identifiers "provide hints about a program's purpose" (Siegmund et al. 2017), which allows experienced programmers to comprehend a program using a top-down approach (Brooks 1983). This has been confirmed by the aforementioned studies that have shown that "commented programs and programs containing full word identifiers are easier to understand" (Arnaoudova et al. 2016).

Over the years, research has been conducted to improve identifier quality, including the development of naming guidelines for identifiers (Butler et al. 2010; Arnaoudova et al. 2016; Hilton and Hermans 2017), and the application of NLP techniques to standardize identifiers (Caprile and Tonella 2000) or even recommend better versions (Lin et al. 2017). More recent approaches have extended this to transformer LMs that automatically rename identifiers in code (Li et al. 2021). Automatically renaming variable names using an LM is relatively straightforward. Generative code LMs, such as GitHub's Copilot (Chen et al. 2021; GitHub 2024), are trained on large collections of open-source repositories and are widely used for code autocompletion, as detailed in Chapter 4. When prompting a (masked) LM with a hidden identifier, multiple versions of the identifier can be generated. Note that to predict an identifier name, it cannot be present in the context (otherwise the model would favor the identifier from the context). So additional masking tokens have to be used to hide other occurrences of the identifier. The SyntaxPT model from Chapter 4 is particularly well-suited for this task, as it has been trained with a multi-task learning strategy that included identifier deobfuscation as a pretraining task. During pretraining the model predicts the most likely name for a hidden identifier by estimating a probability distribution for the next token, given all previously generated tokens and an input sequence (see Equation (2.16)). This model can be used for automatic identifier

renaming by first masking the identifier (and all its occurrences) and then sampling one or more candidates to present to the user (Mastropaolo et al. 2022).

However, blindly renaming all identifiers in a code file is impractical and generally unwelcome by developers. A more useful approach in practice would be to *highlight identifiers of insufficient quality for the developer* that would benefit from refactoring. A model that could detect the quality of an identifier would be a valuable tool for developers, as it could provide feedback on the quality of their identifiers, guide them in writing better identifiers, and improve the overall quality of the source code. Identifiers of insufficient quality could be highlighted in an IDE or in a code review tool to the developer, allowing him/her to either manually refactor the identifiers or use generated candidates from an LM.

Quantifying the quality of an identifier is challenging due to variations in coding styles influenced by personal preferences, company guidelines, and language-specific naming conventions. However, widely accepted coding conventions in the form of identifier naming guidelines (Butler et al. 2010; Arnaoudova et al. 2016; Hilton and Hermans 2017) provide precise rules that developers should follow, such as *"Use words from a dictionary and no uncommon abbreviations"*. Measuring identifier quality by adherence to these guidelines has many advantages, as they are mostly language-agnostic, widely accepted, and most importantly offer a way to measure the quality of identifiers objectively. This is why in this chapter, identifiers that violate naming guidelines will be referred to as *low-quality* or *bad*, while those adhering to the rules are considered *high-quality* or *good*. Therefore, to evaluate how well an LM assesses the quality of an identifier, this chapter quantifies how effectively it detects violations of these guidelines.

7.1.1 Contributions

To this end in a first contribution, this chapter presents and compares four different ways of extracting identifier quality scores from an LM. All these strategies use the SyntaxPT model from Chapter 4, but could be applied to any LM that supports masked language modeling, which is demonstrated by the InCoder and GraphCodeBert models. The approach is visualized in Figure 7.1. To identify low-quality identifiers, the SyntaxPT model from Chapter 4 is used in two different self-supervised strategies to predict identifier quality:

1. **Generative Rating**: SyntaxPT is used to estimate the probability of the identifier, without further fine-tuning, as shown in Figure 7.1a. This is done in three ways: The first two directly use the identifier's likelihood and the last compares its likelihood-ratio to the most likely identifier predicted by the model.

2. **Discriminative Rating**: This strategy is motivated by the fact that the generative model has never learned to discriminate between good and bad identifiers. Identi-

fiers are replaced by alternative versions sampled from a word embedding model, and SyntaxPT's encoder is fine-tuned to discriminate original identifiers from replaced ones.

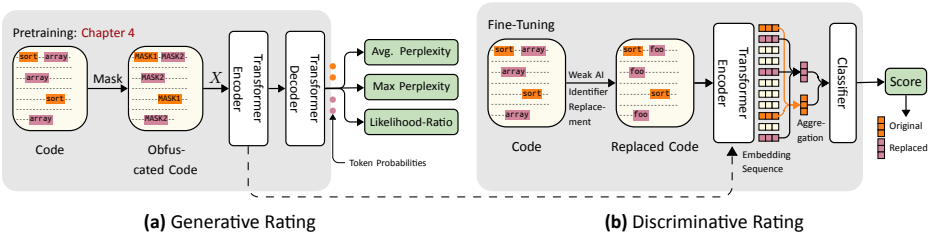


Figure 7.1: The SyntaxPT model—an encoder-decoder transformer LM described in Chapter 4—was pretrained, among other tasks, with identifier deobfuscation. In this chapter it is used to assess the quality of identifiers with two ranking strategies (a) and (b), with four proposed scoring functions (green). All are inherently self-supervised. In (a), the generative strategy uses the LM's likelihoods in a zero-shot setting without further fine-tuning, with three scoring functions: (1 and 2) directly using the identifier's likelihood/perplexity and (3) comparing its likelihood ratio to the most likely identifier predicted by the model. In (b), the discriminative strategy fine-tunes the encoder of SyntaxPT to distinguish between real identifiers and those replaced by a weak AI. Figure adapted from **Villmow*** et al. (2023b) ©2023 IEEE.

Directly using the likelihood of an identifier as an identifier quality score has been explored before with n-grams language models (Allamanis et al. 2014). This chapter builds on this foundation and addresses its open challenges: First, multi-token identifiers pose a problem. The first token typically has a lower probability, because many options are possible. Once it is clear how the identifier starts, the continuation will be "easy" (fewer choices possible), thus the model will assign a higher probability to subsequent tokens. For instance, consider the identifier `number_of_items_in_cart`, that gets easier to predict after the first few tokens. Second, other identifiers in the target identifier's context matter. The model is trained to predict the most likely identifier for a given context, which does not necessarily have to be a high-quality one. For example, in code where all other identifiers are single-letter abbreviations, the model may assign a high probability to a poor-quality identifier (i.e., a single-letter abbreviation), while assigning a low score to a high-quality identifier. To address these issues, this chapter proposes and compares novel alternative scoring functions and discusses the impact of masking on generative scoring functions.

It can be hypothesized that by predicting the most likely identifier for a given context, the SyntaxPT model implicitly learns what makes a good identifier. Assuming that most code in open-source repositories implements best practices in software development and contains high-quality identifiers, the LM should predict high-quality identifiers more often than low-quality identifiers. This chapter aims to test this hypothesis by evaluating the performance of the LM on a dataset of identifiers labeled with respect to their adherence to identifier naming guidelines. Only few datasets consider identifier quality. For example,

Chen et al. (2022) build a dataset for measuring the similarity between identifiers, by automatically collecting two versions of an identifier from GitHub commits, assuming that the newer version is of better quality. However, this can be a noisy measure, and the authors do not consider naming guidelines. To the best of the author's knowledge, no dataset with human-labeled quality assessments for identifiers exists, particularly none regarding coding guideline violations.

Therefore, this chapter addresses this research gap by introducing the first benchmark for detecting violations of identifier naming guidelines. The dataset contains 6,203 dense annotations of identifiers across 28 common naming guidelines, organized into four groups: syntax, vocabulary, data type, and method name. The dataset has been made publically available to support further research (**Villmow** et al. 2023a). On this dataset, the proposed scoring functions are evaluated and compared.

In summary the contributions of this chapter are:

- The first dataset for assessing the quality of identifiers, which is based on established coding guidelines (**Villmow** et al. 2023a).
- Propose and evaluate three distinct generative and one discriminative scoring functions to extract identifier quality scores from a LM. All scoring functions are self-supervised. Most scoring functions are novel itself (to the author's knowledge), but especially the application to identifier quality estimation with respect to naming guidelines is novel and has not been previously explored. The generative scoring functions can be applied to any masked LM and are designed to facilitate comparison with future research that may employ more complex methods. Additionally, the impact of masking on the generative scoring functions is discussed.
- Compare the SyntaxPT model against other state-of-the-art LMs on this task and dataset and demonstrate that it outperforms even the larger InCoder (1.3B parameter) model by a large margin.
- Provide an in-depth analysis of guideline-specific performance of the generative and discriminative scoring functions, and discuss what makes a guideline hard to detect.

7.2 Related Work

The importance of identifiers in source code has long been recognized in software engineering literature. Identifiers typically constitute up to 70% of the code (Deissenboeck and Pizka 2006), and their quality directly affects program comprehension and maintainability (Lawrie et al. 2006; Takang et al. 1996; Fakhoury et al. 2020). In this section, prior research on the role of identifier quality in code comprehension, automatic renaming

approaches, and the application of machine learning models, particularly language models, to code and identifier quality assessment are reviewed. For a more comprehensive review please refer to Li et al. (2021).

7.2.1 Impact of Identifier Naming on Code Comprehension

Identifiers are important for embedding concepts in communication between developers, and the choice of identifier names influences the code's readability and maintainability (Fakhoury et al. 2020). Comprehending a program plays a central role in many software engineering tasks such as code reviews, debugging, and maintenance (Brooks 1983). As code readability strongly influences comprehension time and effort, identifiers, which serve as semantic beacons, can either ease or hinder the understanding of source code (Siegmund et al. 2017). Siegmund et al. (2017) demonstrated that well-chosen identifiers significantly lower the cognitive load of developers by providing meaningful cues about a program's purpose.

Studies such as those by Gellenbeck and Cook (1991) and Lawrie et al. (2006) further confirm that full, meaningful names are more comprehensible than abbreviated or non-intention-revealing ones. Takang et al. (1996) empirically evaluated the combined effect of identifier naming and comments on comprehension, and conclude that poor-quality identifiers complicate code understanding. Cognitive load theory posits that high cognitive load negatively affects performance and increases the likelihood of errors (Sweller 2011; Chen et al. 2016), which is also the case when developers are faced with poor-quality identifiers (Fritz et al. 2014; Andaloussi et al. 2022). Empirical studies using brain imaging techniques such as functional Near Infrared Spectroscopy (fNIRS) and functional Magnetic Resonance Imaging (fMRI) have confirmed that lexical inconsistencies and low-quality identifiers significantly increase developers' cognitive load (Fakhoury et al. 2020; Siegmund et al. 2017).

7.2.2 Naming Guidelines

To address the challenges posed by low-quality identifiers, researchers have proposed naming guidelines to help developers write better identifiers (Butler et al. 2010; Arnaoudova et al. 2016; Hilton and Hermans 2017). These guidelines provide precise rules for constructing identifiers, including syntax guidelines (how identifiers are formatted), vocabulary guidelines (word choice), data type guidelines (including data type names in identifiers), and method name guidelines (Hilton and Hermans 2017). Studies have demonstrated that adherence to naming guidelines improves code comprehension (Lawrie et al. 2006; Takang et al. 1996). For example, Lawrie et al. (2006) found that full identifier names are more comprehensible than abbreviated ones. Similarly, Takang et al. (1996) measured the combined effect of identifier naming and comments on developers' ability to understand code, concluding that meaningful identifiers significantly aid comprehension.

While various readability metrics have been proposed (Buse and Weimer 2010), studies on identifier quality have emphasized that readability cannot be solely captured by syntactic measures. Butler et al. (2010) find that "the multifactorial nature of identifier quality makes measurement problematic". This chapter takes a step towards addressing this challenge by providing a dataset of identifiers annotated with respect to adherence to the aforementioned guidelines.

7.2.3 Automatic Improvement of Identifier Names

Given the importance of high-quality identifiers, various approaches have been proposed to automatically improve identifier names. These can be broadly categorized into rule-based and learning-based methods.

Rule-Based Approaches

Early rule-based approaches such as those by Caprile and Tonella (2000) standardized identifier names by imposing lexicon constraints and term sequencing rules. Similarly, Deissenboeck and Pizka (2006) proposed a formal model based on bijective mappings between concepts and names. Later works explored the harmonization of identifier names through an analysis of variable assignments (Thies and Roth 2010), and tools like CREN (Jablonski and Hou 2007) and Smart Formatter (Corbo et al. 2007) applied coding style rules learned from existing source code to code in an IDE. However, these techniques are often limited to specific coding styles or do not generalize well to different contexts.

Learning-Based Approaches

More recent approaches use data-driven machine learning, particularly language models, to improve identifier names. Hindle et al. (2012) introduced the concept of the *naturalness* of software and found that code is repetitive and predictable, even more than natural language. Allamanis et al. (2018) found "software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools". For example, NATURALIZE (Allamanis et al. 2014) used a n-gram language model to detect and suggest corrections for surprising or inconsistent identifier names. The authors use a language model's likelihood as a surprise metric to detect refactoring opportunities, which is similar to the approach in this chapter. However, they don't test the adherence of that score to naming guidelines. Other work has applied n-gram language models to detect anomalies in code, and point out that improbable code may indicate bugs (Ray et al. 2016), or by refining the NATURALIZE model to produce more consistent identifiers (Lin et al. 2017). However, Lin et al. (2017) found that the aforementioned approaches to identifier renaming still generate a high number of false positives, which may hinder their practical application and emphasizes the need for a better indicator of identifier quality.

While these early language models were primarily n-gram-based, more recent work has shifted towards deep learning approaches. Context2Name (Bavishi et al. 2018) and DIRE (Lacomis et al. 2019) applied neural networks to infer meaningful variable names in decompiled code. However, these models mainly focus on recovering meaningful names from obfuscated or unclear contexts, rather than assessing adherence to established naming guidelines. Other works have focused on method naming (which has been detailed in Section 3.2), e.g, Allamanis et al. (2015) introduced a neural context model for suggesting accurate method and class names. Liu et al. (2022) proposed *GTNM*, a transformer-based model that considers local context, project-specific context, and documentation to recommend method names.

Transformer Models for Identifier Quality and Naming The rise of transformer-based LMs has significantly advanced the field of code understanding and generation (see Section 4.2). Large pretrained models such as SyntaxPT from Chapter 4, CodeBert, CodeT5, and InCoder (Fried et al. 2023) have been applied to various software engineering tasks (Ciniselli et al. 2022), including identifier renaming. For example, Mastropaolo et al. (2023) evaluated transformer LMs for automatic variable renaming and demonstrated their potential for detecting and refactoring variable names. However, they primarily assessed the correctness of the renamed identifier by comparing it to refactored names in GitHub repositories, without investigating how well the models conformed to coding guidelines. Their work offers limited insight into *why* identifiers were refactored and which properties make bad identifiers easy or difficult to spot, which is addressed in Section 7.6.3. Chen et al. (2022) proposed VarCLR, a method that learns semantic representations of variables via contrastive learning, to capture the similarity between identifiers. However, the authors encoded variables without additional context, which may limit the model's ability to capture the full meaning of an identifier. Similar to this chapter, Sengamedu and Zhao (2022) evaluated the probabilities from LMs for general code quality identification, but focused on issues such as token-level errors or unnatural code. It is similar in the use of a LM's likelihood or perplexity to detect code quality problems, but the authors did not specifically evaluate adherence to coding conventions or identifier naming guidelines. Evaluating the likelihood on a sub-token level of a decoder model is not directly comparable to the approach of this chapter in which *all* occurrences of an identifier are masked. Their approach rather detects variable misuses, where the wrong variable is used at a certain position. This is something our approach does not consider.

As argued above this chapter extends prior research by directly addressing the challenge of evaluating identifier quality based on adherence to established naming guidelines. While previous studies have demonstrated the potential of using language models for identifier renaming or assessing code quality (Mastropaolo et al. 2023; Hindle et al. 2012), none

have systematically examined how well language models detect violations of naming conventions.

7.3 Approach

This section describes the approach to analyze the quality of identifiers in code using the SyntaxPT LM from Chapter 4 in two self-supervised strategies. Recap, that the model first tokenizes code into input tokens $\boldsymbol{c} = (c^{(1)}, \ldots, c^{(n)})$ using the syntax-tree aware BPE (see Section 4.5.4). This allows to utilize the syntax tree to detect all occurrences of an identifier. Each identifier may not only consist of a single token, but rather a multi-token subsequence of $\boldsymbol{c}$, since the BPE tokenizer may split it into multiple tokens. So an identifier is represented by a sequence of tokens, that can appear multiple times in $\boldsymbol{c}$. Let $\mathbb{I}$ denote the set of all identifiers appearing in $\boldsymbol{c}$.

The model has been pretrained with a multi-task denoising objective, in which subsequences $\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_k$ of $\boldsymbol{c}$, such as identifiers, have been replaced by sentinel tokens to form a noised input sequence $\boldsymbol{x}$, defined in Equation (4.3). The input sequence $\boldsymbol{x}$ is fed into the encoder, which produces an intermediate output sequence of embeddings $\boldsymbol{z}$. Based on these embeddings, the decoder produces an output sequence $\boldsymbol{y} = (x^{(mask_1)}) \oplus \boldsymbol{Y}_1 \oplus \ldots \oplus (x^{(mask_k)}) \oplus \boldsymbol{Y}_k \oplus (x^{(eos)})$ (see Equation (4.4)). Thereby, the model autoregressively predicts the likelihood $p(y^{(i)} \mid \boldsymbol{y}^{(<i)}, \boldsymbol{x})$ of each token $y^{(i)}$ of the target sequence $\boldsymbol{y}$. In the case of identifier deobfuscation, the model predicts the likelihood of each token of an identifier. The model is trained by optimizing its parameters $\boldsymbol{\theta}$ to maximize the following objective function:

$$\log \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \log p(y^{(i)} \mid \boldsymbol{y}^{(<i)}, \boldsymbol{x}) \tag{7.1}$$

Note that this is Equation (2.16) slightly rewritten to be maximizing log probability instead of minimizing the negative log-likelihood loss.

Given this pretrained model that is able to predict identifier names, this chapter proposes two different strategies to use the model to estimate the quality of an identifier:

1. The generative rating, visualized in Figure 7.1a, uses the likelihoods of the LM directly,

2. The discriminative rating, shown in Figure 7.1b, fine-tunes a classifier that uses the encoder output $\boldsymbol{z}$ to distinguish between real and fake identifiers inserted from a FastText model (Bojanowski et al. 2017). This model is called $\text{SyntaxPT}_{\text{discriminative}}$ from now on.

Both strategies are self-supervised and therefore do not require labeled training data. The details of each approach will be discussed in Sections 7.3.1 and 7.3.3, with a probabilistic analysis provided in Section 7.3.2.

7.3.1 Generative Rating

Given that the model has been trained to deobfuscate identifiers, the likelihood of an identifier within a given context can be directly computed from the model's output probabilities. Let $\mathbb{I}' \subseteq \mathbb{I}$ denote the subset of all identifiers that will be obfuscated in $\boldsymbol{x}$. Consider identifier $I^{(i)} \in \mathbb{I}'$ with tokens $\boldsymbol{Y}_i$ representing the subsequence $(y^{(j)}, \ldots, y^{(j')})$ of $\boldsymbol{y}$. In this chapter, the *log-likelihood of identifier* I^i is defined as the average log-likelihood of its tokens in the output sequence:

$$\log \mathrm{L}_{\boldsymbol{\theta}}(\boldsymbol{Y}_i, \boldsymbol{x}) = \frac{1}{j' - j + 1} \sum_{k=j}^{j'} \log p(y^{(k)} \mid \boldsymbol{y}^{(<k)}, \boldsymbol{x}) \tag{7.2}$$

Given this definition of the log-likelihood of an identifier one can see that it depends on two major factors which define how much context is available to the model when predicting an identifier: the construction of (1) the input sequence $\boldsymbol{x}$, and (2) the output sequence $\boldsymbol{y}$. First the *total number of hidden identifiers* in $\boldsymbol{x}$ strongly impacts the log-likelihood of an identifier. The more identifiers are obfuscated, the less information is available to the model from other identifiers, and the less confident are its predictions. Moreover, the position of an identifier in the output sequence influences its likelihood, since the autoregressive transformer decoder can attend to previously generated tokens. During deobfuscation, the model can use identifiers $I^{(<i)}$ as additional context when predicting identifier $I^{(i)}$. This can lead to higher log-likelihoods for identifiers that occur later in the sequence.

Scores

This section present three scoring functions that utilize the log-likelihood of an identifier to estimate its quality. Subsequently, two different masking strategies will be introduced to address the aforementioned context factors. Given that code quality analysis should be reasonably fast, performance considerations are essential when developing these scoring functions. Thus, the computational requirements for each method will be discussed.

Perplexity Perplexity is a common measure used to evaluate LMs. Jurafsky and Martin (2009) define the perplexity of an LM on a test set as the inverse probability of the test set, normalized by the number of words. Due to the inverse relationship between perplexity and likelihood, higher perplexity indicates lower probabilities for the individual tokens. From an information theory perspective, it is closely related to entropy

and measures how "surprised" the model is by the test set, or, in other words, the degree of uncertainty the model has in its predictions. Perplexity has the advantage of being independent of the length of the given sequence.

With the hypothesis that low-quality identifiers are less likely, perplexity can be used to directly assess the quality of an identifier I^i. The *perplexity-score* is defined as:

$$\text{Perplexity}(I^i) = \exp(-\log \text{L}_\theta(\boldsymbol{Y}^i, \boldsymbol{x})) \tag{7.3}$$

When this score is high for an identifier, it indicates that the model is surprised by the identifier, and it is likely to be of low quality. This perplexity-score requires only a single forward pass through the model.

Max-token Perplexity With the intuition that a single suspicious part of an identifier can make the whole identifier bad, another generative scoring function is introduced that computes the perplexity of each token in the identifier individually and returns the maximum perplexity of all tokens in the identifier. The *max-token perplexity* is defined as:

$$\text{MaxTokenPerplexity}(I^i) = \max_{k \in \{j,\dots,j'\}} \exp(-\log p(y^{(k)} \mid \boldsymbol{y}^{(<k)}, \boldsymbol{x})) \tag{7.4}$$

Recap that the identifier I^i is represented by the subsequence $(y^{(j)}, \dots, y^{(j')})$ of $\boldsymbol{y}$.

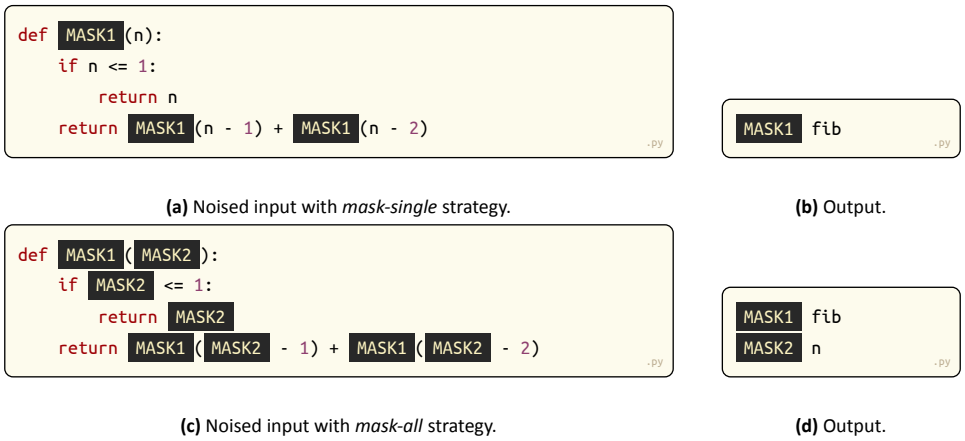
Log-likelihood Ratio In complex contexts, the model might distribute the probability mass among a large set of possible identifiers. In such cases, high perplexity or low probability of an identifier does not necessarily indicate low quality, as many other identifiers may be equally (un)likely. These scenarios can be detected by analyzing whether the LM finds another identifier much more plausible, in the place of $I^{(i)}$, the identifier to score. This analysis is performed in a third scoring function: first, the most likely identifier in-place of $I^{(i)}$ is generated[1], then the log-likelihood ratio of the existing identifier compared to the best one is analyzed.

Formally, when quantifying identifier $I^{(i)}$'s tokens $\boldsymbol{Y}_i$, and $\hat{\boldsymbol{Y}}_i$ is the best identifier predicted by the model in place of $\boldsymbol{Y}_i$, the *log-likelihood ratio score* is defined as:

$$\text{LikelihoodRatio}(I^i) = \frac{\log \text{L}_\theta(\hat{\boldsymbol{Y}}_i, \boldsymbol{x})}{\log \text{L}_\theta(\boldsymbol{Y}_i, \boldsymbol{x})} \tag{7.5}$$

This score is high when the identifier is suspicious. However, it is computationally more demanding than the perplexity scores since it requires to first generate the most likely

[1] Beam search with width 5 is used when generating identifiers.



(a) Noised input with *mask-single* strategy.

(b) Output.

(c) Noised input with *mask-all* strategy.

(d) Output.

Figure 7.2: Example of identifier deobfuscation with different masking strategies. In (c) the *mask-all* strategy is used, where every identifier is obfuscated in the context. The model is trained to predict the concatenated output sequence with the original identifiers. Note that in this setting when predicting identifier `fib` the identifier `n` is also hidden, but when predicting `n` identifier `fib` is visible to the model. In (a) the *mask-single* strategy is used, in which only a single identifier is obfuscated in the context. To score all identifiers this strategy is applied to each identifier individually.

identifier—which itself requires multiple forward passes through the model—and then another forward pass to compute the log-likelihood of the given identifier.

Masking Strategies

As outlined above, the amount of masking in the context and the order of identifiers in the output sequence are two factors that strongly influence the log-likelihood of an identifier. To address these factors, two separate masking strategies are explored in this section, both of which can be used in combination with all of the above scoring functions. The masking strategies are visualized in Figure 7.2 and described in the following. Table 7.1 lists the requirements for all combinations of scoring and masking strategies.

Mask-Single The maximum amount of contextual information is provided to the model when only a single identifier is obfuscated, i.e., $|\mathbb{I}'| = 1$. To compute a score for every identifier in $\mathbb{I}$ using this masking strategy, the scoring method must be applied multiple times—once for each identifier. This approach allows the scoring function to exploit other identifiers to draw conclusions about the hidden identifier.

	Perplexity		Likelihood Ratio	
	F	**G**	**F**	**G**
Mask-All	1	0	1	n
Mask-Single	n	0	n	n

Table 7.1: Number of single forward passes (F) and generations (G) necessary for each combination of scoring and masking method for a file with n identifiers, i.e. $n = |\mathbb{I}|$.

However, using contextual information may introduce an unwanted bias into the model when estimating identifier quality. For example, when masking an identifier in a piece of code with low-quality identifiers, where all identifiers are single characters, the most likely identifier for the LM is also a single-character identifier. The goal is to estimate the quality of an identifier, not merely its goodness-of-fit with other identifiers in its context. A countermeasure is to reduce the amount of contextual information available to the model.

Mask-All This is explored in the *mask-all* method, in which all available identifiers are hidden in $\boldsymbol{x}$, making $\mathbb{I}' = \mathbb{I}$. Since the model can only attend to identifiers that appear earlier in the output sequence, as previously outlined, this approach provides limited contextual information to the model. However, it is also the fastest masking scheme, as the log-likelihood for every hidden identifier can be computed in a single forward pass.

7.3.2 Probabilistic Interpretation

The goal with identifier assessment is to develop a detector for low-quality variables, which frames the task as a binary classification problem. Given supervised training data of code contexts $\boldsymbol{x}$, in which identifiers have been annotated whether they are of low-quality and belong to class $C{=}1$, or are of high-quality and belong to class $C{=}0$, a binary classifier can be trained to predict the class of an identifier I given its context $\boldsymbol{x}$. This corresponds to estimating the conditional probability $P(C{=}1 \mid I, \boldsymbol{x})$.

Note that the generative approach introduced above estimates the overall probability of an identifier given an obfuscated context $P(I \mid \boldsymbol{x})$ (compare Equation (7.2)). To examine this from a probabilistic viewpoint, the posterior probability can be expressed as follows:

$$
\begin{aligned}
P(C{=}1 \mid I, \boldsymbol{x}) &= \frac{P(C{=}1, I, \boldsymbol{x})}{P(C{=}0, I, \boldsymbol{x}) + P(C{=}1, I, \boldsymbol{x})} \\
&= \frac{\overbrace{P(C{=}1)}^{(7.6a)} \cdot \overbrace{P(\boldsymbol{x} \mid C{=}1)}^{(7.6b)} \cdot \overbrace{P(I \mid \boldsymbol{x}, C{=}1)}^{(7.6c)}}{\underbrace{P(C{=}0)}_{(7.6d)} \cdot \underbrace{P(\boldsymbol{x} \mid C{=}0)}_{(7.6e)} \cdot \underbrace{P(I \mid \boldsymbol{x}, C{=}0)}_{(7.6f)} + \underbrace{P(C{=}1, I, \boldsymbol{x})}_{(7.6g)}}
\end{aligned}
\tag{7.6}
$$

The individual terms in the equation above have the following interpretations:

- $P(C{=}1)$ (Term 7.6a) is the prior probability that an identifier is low-quality. It is a constant that reflects the distribution of low-quality identifiers in source code.
- $P(\boldsymbol{x} \mid C{=}1)$ (Term 7.6b) is the probability of code context $\boldsymbol{x}$ occurring, given that the identifier is low-quality. For example, poor identifiers are more likely to appear in poorly written code contexts.

- $P(I \mid \boldsymbol{x}, C{=}1)$ (Term 7.6c) is the likelihood of an identifier I given the context $\boldsymbol{x}$, assuming the identifier is low-quality. The generative approach lacks this term, as it presumes all code in the training phase to be of high quality.
- $P(C{=}0)$ (Term 7.6d) is the prior probability that an identifier is high-quality.
- $P(\boldsymbol{x} \mid C{=}0)$ (Term 7.6e) is the probability of a given code context $\boldsymbol{x}$ occurring, given that the identifier is high-quality.
- $P(I \mid \boldsymbol{x}, C{=}0)$ (Term 7.6f) is approximated by $P(I \mid \boldsymbol{x})$ in the generative approach, as the assumption is made that all code in pretraining is high-quality. However, this is a poor approximation, as not all identifiers in the training data are necessarily of high quality (training data is crawled from GitHub at a large-scale).
- Term (7.6g) is same as product of Terms (7.6a) to (7.6c).

These terms have three major factors, which affect the probability of an identifier being low-quality, which are not directly captured in the generative approach:

1. The generative approach assumes that all code in the pretraining phase is high-quality (Term 7.6f), since $P(I \mid \boldsymbol{x}, C{=}0) \approx P(I \mid \boldsymbol{x})$. This assumption may not hold for arbitrary software projects. To ensure this approximation is as accurate as possible, it is necessary to select high-quality software projects for pretraining. Recap that during the collection of the pretraining dataset in Section 4.5.5, quality filters were used to ensure that only projects with active development and a high number of stars on GitHub are selected. Only projects that passed these criteria were used for pretraining the generative model.
2. Terms (7.6b) and (7.6e) suggest that low-quality identifiers are more likely to appear in poor code contexts, and high-quality identifiers in well-written code contexts. For instance, in a code context where all identifiers are single characters, the likelihood of an identifier being low-quality is higher than in a context where all identifiers are meaningful words. This creates a challenge for the generative approach, as it conflates identifier quality with its contextual likelihood. To mitigate this, different masking strategies, such as mask-single and mask-all (see Section 7.3.1), are explored in this chapter to reduce the amount of contextual information available when computing the log-likelihood of an identifier.
3. Term (7.6c) highlights that the generative approach does not model the counterclass, i.e., it does not discriminate between good and bad identifiers. This is an inherent limitation of the generative approach, which the next section will address by introducing a discriminative approach.

7.3.3 Discriminative Rating

As discussed above the generative approach assumes that all code in the training data is of high quality and lacks a counterclass model for poor identifiers. During pretraining, the model does not learn to differentiate between good and bad identifiers. The generative scoring function computes the likelihood of an identifier given a certain context, which works well when bad identifiers are unlikely. However, this approach may not always yield the desired results, and a model that estimates the counterclass ($C{=}1$) directly can be preferable.

To address this issue and to develop a model that estimates the posterior $P(C{=}1 \mid I, \boldsymbol{x})$ directly, this section introduces a discriminative approach. An encoder is trained on a binary classification problem to achieve this goal. Given the scarcity of annotated data for good and bad identifiers, the model is trained in a self-supervised manner. A weaker LM (FastText) is used to sample weaker but plausible versions of identifiers, similar to discriminator language models such as Electra (Clark et al. 2020) in NLP and CodeBert, which are trained to detect replaced tokens (see Section 4.2). However, in the proposed discriminative approach full identifiers are replaced by supposedly worse but still realistic versions, instead of replacing individual tokens.

An unsupervised word embedding (FastText) is trained on the identifiers in the training data (Bojanowski et al. 2017). For each real identifier I, alternative identifiers are sampled from the top five most similar identifiers, with the probability of selection proportional to the cosine similarity between the FastText embeddings. For instance, `TRAIL_RANGE` might be replaced with `RANGE`, `ALLOWABLE_ERROR` with `ERROR_CODE`, or `setModel` with `setRepeatMode`. Details on dataset creation are provided in Section 7.4.

Once the FastText model has been trained, the approach operates similarly to the generative approach, but instead of replacing identifier subsequences $\boldsymbol{Y}_i$ with mask tokens to create a noised context $\boldsymbol{x}$, the context is created by replacing identifier subsequences with other identifiers $\boldsymbol{Y}_i'$. Let $\mathbb{I}$ denote the set of identifiers in $\boldsymbol{x}$, where some are original and others have been replaced by the FastText model. After processing the noised sequence through the transformer encoder to obtain output states $\boldsymbol{z} \in \mathbb{R}^{n \times d}$, a single contextualized embedding $\boldsymbol{s}^{(i)} \in \mathbb{R}^d$ for every identifier $I^{(i)} \in \mathbb{I}$ is computed the following way: Since identifier $I^{(i)}$ may consist of multiple tokens and occur at several positions in the noised input. To aggregate the individual token representations corresponding to $I^{(i)}$, max-pooling is employed (Goodfellow et al. 2016). Let $\mathbb{J}^{(i)}$ denote the set of token indices corresponding to the identifier $I^{(i)}$. The identifier embedding is then computed as

$$\boldsymbol{s}^{(i)} = \max_{j \in \mathbb{J}^{(i)}} (\boldsymbol{z}^{(j)}) \tag{7.7}$$

Dataset	Train	Validation	Test
Pretraining Dataset	32M	50.000	-
Fine-Tuning Dataset	64,474	13,911	13,854
Manually Annotated Dataset	-	-	1,770

Table 7.2: The number of code files in the training, validation, and test sets for each dataset used in this chapter.

where the max operation is applied element-wise across all dimensions of the token representation vectors $\boldsymbol{z}^{(j)}$. Max-pooling is used here because the quality determination of an identifier from its individual tokens is invariant to their positions, and the strongest signal indicating a suspicious identifier is desired.

Next, the identifier embedding $\boldsymbol{s}^{(i)}$ is fed into a binary classification head that estimates $P(C{=}1 \mid I, \boldsymbol{x})$. Following common practice the classification head projects the identifier embedding to a scalar probability using two linear layers, with a ReLU activation function in between and a sigmoid activation function at the end:

$$P(C{=}1 \mid I^{(i)}, \boldsymbol{x}) = \sigma(\mathrm{ReLU}(\boldsymbol{s}^{(i)} \cdot \boldsymbol{A}) \cdot \boldsymbol{B}) \tag{7.8}$$

Here, $\boldsymbol{A} \in \mathbb{R}^{d \times d}$ and $\boldsymbol{B} \in \mathbb{R}^{d \times 1}$ are the weight matrices of the two linear layers, and σ is the sigmoid function. The model is trained using the binary cross-entropy loss:

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right) \tag{7.9}$$

where N is the number of identifiers in the batch, $y^{(i)}$ is the ground truth label for identifier no. i, and $\hat{y}^{(i)} = P(C{=}1 \mid I^{(i)}, \boldsymbol{x})$ is the predicted probability that the identifier is bad.

7.4 Datasets

The pretraining dataset used for training the generative model is described in Section 4.5.5. This section introduces two new datasets:

1. The *fine-tuning dataset* for training the discriminative models. This dataset consists of Java files in which identifiers have been partially replaced by a weak AI.

2. A smaller, *manually-annotated evaluation dataset* has been constructed to increase confidence in the quality of the considered identifiers. In this dataset, identifier quality was manually inspected and matched against coding guidelines.

Precise statistics of all datasets used in this chapter are provided in Table 7.2.

7.4.1 Fine-Tuning Dataset

To fine-tune a transformer encoder model to discriminate between good and bad identifiers, a dataset was constructed in which identifiers were replaced with semantically similar but less meaningful substitutes sampled from a weaker AI, as outlined above. This section describes the construction of this fine-tuning dataset for the discriminative model.

FastText Model

A good candidate for a weak AI is to use static word/identifier embeddings (Mikolov et al. 2013b). These are trained unsupervised on large amounts of text and have been frequently used in NLP until the rise of contextualized transformers. Given a word embedding model, similar identifiers can be retrieved by performing a nearest neighbor search based on the cosine similarity between their embeddings. In this chapter the FastText word embedding model is used, since it incorporates subword information to obtain an embedding for words not seen during training (Bojanowski et al. 2017). This is particularly useful for identifiers, given that they are often composed of multiple individual words.

```
1 [...] height toBlocking first response getResult response getResult getMiners Exception e
  ↪ log error e getMessage e Collections emptyList [...]
2 [...] sendResult rocketMQTemplate syncSend UserSmsCodeSaveMessage TOPIC message
  ↪ SendStatus SEND_OK equals sendResult getSendStatus log error [...]
3 [...] member setPassword pwd member setCreateTime now member setUpdateTime now member
  ↪ setChannelId loginWay member setPlatform MemberConstant [...]
                                                                                    .java
```

Figure 7.3: For the FastText training, the files were preprocessed to contain only their identifiers, separated by spaces each code file becomes a line.

To train the word embedding model FastText's released software package (Joulin et al. 2016) was used. The training set was constructed from Java files of the validation set of the pretraining dataset (see Section 4.5.5). In each code file all non-identifier tokens were discarded, and the remaining identifier mentions separated by spaces, as shown in Figure 7.3. In total, the training data for the word embedding consisted of 1,809,110 identifier mentions. The model was trained for five epochs using the CBOW approach with a learning rate of 0.01, an embedding dimensionality of d=100, a context window of size 5, and minimum and maximum subword sizes set to 3 and 6, respectively.

After training, the model stores word embeddings for every identifier seen during training in a matrix $\boldsymbol{E} \in \mathbb{R}^{|\mathbb{V}| \times d}$, where $\mathbb{V}$ is the set of unique identifiers in the training data and d is the embedding dimensionality. Additionally, the model can provide embeddings for unseen identifiers by averaging the embeddings of their subwords. This matrix is normalized to unit length along the embedding dimension, so that the cosine similarity

between two embeddings $\boldsymbol{E}_i$ and $\boldsymbol{E}_j$ can be computed as their scalar product. Here, the embedding at index i in the embedding matrix $\boldsymbol{E}$ belongs to the identifier $\mathbb{V}^{(i)}$.

Dataset Construction

Given the FastText model, the fine-tuning dataset of approximately 92k Java files was created. The files used for this dataset have not been in the training set of the pretraining dataset (see Section 4.5.5 for details about the pretraining dataset). Given a code file, a subset of its identifiers $\mathbb{I}' \subset \mathbb{I}$ was selected to be replaced by similar identifiers, so that $|\mathbb{I}'| \leq 0.2 \cdot |\mathbb{I}|$. The replacement I' for each identifier $I \in \mathbb{I}'$ was sampled from the top five most similar identifiers to I according to the FastText model.

Formally, let $\boldsymbol{w} \in \mathbb{R}^d$ be the normalized embedding of the existing identifier I, which does not necessarily have to be in the FastText training data and thus in $\boldsymbol{E}$, since FastText is a subword embedding model. The cosine similarity between $\boldsymbol{w}$ and all embeddings in $\boldsymbol{E}$ is computed, and the indices of the top five most similar identifiers are selected:

$$\arg\text{top5}_{i \in \{1,\ldots,|\mathbb{V}|\}} \left(\boldsymbol{E}_{i,:} \cdot \boldsymbol{w}\right) \tag{7.10}$$

Then one of these five identifier indices is sampled with a probability proportional to the cosine similarity, $P(i) \propto \boldsymbol{E}_{i,:} \cdot \boldsymbol{w}$. Let i' be the sampled index, then the replacement identifier $I' = \mathbb{V}^{(i')}$. All occurrences of I in the code file are replaced by I'. This process is repeated for each identifier in $\mathbb{I}'$.

For example, with the FastText model the identifier `user` may be replaced with `users`, `saveUser`, or `User`. Key is the idea that the replaced identifiers are of lower, but sufficient quality to be plausible in real-world code. Note that the replacement `users` directly violates the guideline 15 of using singular nouns for variable names, while `saveUser` may be misleading in the given context (no user is saved, but handled differently) and thus violates guideline 11. To validate that the identifiers sampled from FastText are worse than the real identifiers, but at the same time more plausible than assuming a uniform distribution, a small blind annotation study was conducted that assessed the quality of the replacements. A test person was asked to compare 100 real and sampled FastText identifier pairs without knowing which was the original and which the replacement. The test person found the replacements to be worse in 86% of cases, comparable in 12% of cases, and better in 2% of cases. The same study was conducted for random replacements without FastText. Here the test person found the replacements to be worse in 99% of cases and comparable in 1% of cases. This indicates that the FastText model provides plausible but weaker replacements for identifiers.

7.4.2 Manually Annotated Dataset

The manually annotated test dataset was constructed to evaluate models that assess the quality of the identifiers based on ground truth derived from coding guidelines. Annotators were given a code file, a subset of identifiers occurring in the code file, and a random guideline for assessing the quality of the identifiers. They were asked to either correct the identifier if it did not meet the guideline or to confirm that the identifier was correct and create a plausible but violating version if possible. This process of creating the dataset is now described in detail.

Data Collection

Project	Version	Project	Version
elasticsearch	8.3.3	clojure	1.12.0
Android-Universal-Image-Loader	1.9.5	dropwizard	2.1.1
libgdx	1.11.0	okhttp	5.0.0
storm	2.4.0	presto	0.274
zxing	3.5.0	metrics	4.2.10
netty	4.1.79	RxJava	3.1.5
bigbluebutton	2.5.4	spring-boot	2.7.2
junit	4.13.2	Bukkit	1.7.9
retrofit	2.9.0	nokogiri	1.13.8

Table 7.3: The software projects and versions used in the manually annotated dataset, from which 59 Java files were randomly selected and annotated.

First, a collection of Java code files was selected from 18 top active GitHub projects that spanned various domains, measured by the number of watchers, forks and collaborators. The complete list of projects including the specific versions is shown in Table 7.3. The files should be self-contained and be able to be understood without knowledge of the entire project, and could be comprehended in a reasonable amount of time, given that during annotation all identifiers in the code files were considered. Therefore, a set of rules was applied to select the files: A code file should have

- no more than 5 imports from the Java Class Library,
- no more than 5 imports from external sources,
- at least one function with a minimum of 10 lines to exclude trivial data classes,
- between 100 and 250 LOC.

Apart from these rules the files were chosen randomly from the projects. After the selection, the files were manually shortened to ≈90 LOC by removing comments, empty lines, and irrelevant or hard to understand segments.

ID	Description	Confirming Example	Violating Example	
1	Apply standard case with rigorous consistency.	`databaseName`	`databaseNAME`	[1, 2]
2	Use dictionary words and no (uncommon) abbreviations.	`databaseName`	`dbNm`	[1, 2]
3	Expand single-letter names (except for control variables).	`databaseName`	`d`	[1, 2]
4	Only use one underscore at a time.	`database_name`	`database__name`	[2]
5	Only use underscores between words.	`database_name`	`_databaseName`	[1, 2]
6	Name constant values.	`BOILINGPOINT`	`ONEHUNDRED`	[2]
7	Limit name character length to 20.	`databaseName`	`databaseIdentificationName`	[2]
8	Limit name word count to four.	`databaseName`	`newDatabaseNameOfStudent`	[1, 2]
9	Qualify values with suffixes.	`studentCount`	`countStudent`	[2]

(a) Syntax Guidelines

ID	Description	Confirming Example	Violating Example	
10	Use a descriptive name that conveys a recognizable concept.	`databaseName`	`foo`	[2]
11	Be precise by identifing specific information and purpose.	`databaseName`	`name`	[2]
12	Use standard language, avoid humor and abiguity.	`terminate`	`whack`	[1, 2]
13	Use a large vocabulary: replace phrases with specific terms.	`learningPerson`	`student`	[2]
14	Don't use prefixes or suffixes that encode the data type.	`databaseNameString`	`databaseName`	[2]

(b) Vocabulary Guidelines

ID	Description	Confirming Example	Violating Example	
15	Use singular names for values.	`User user;`	`User users;`	[2]
16	Use plural names for collections.	`User[] users;`	`User[] user;`	[2]
17	Use Boolean variable names that imply true or false.	`isFinished`	`finish`	[2]
18	Use positive Boolean names.	`isFinished`	`notRunningAnymore`	[2]
19	Attribute name and type should be consistent.	`int studentCount;`	`String studentCount;`	[2, 3]

(c) Data Type Guidelines

ID	Description	Confirming Example	Violating Example	
20	Use a verb-phrase name.	`createUser(): User`	`userCreation(): User`	[2]
21	Don't use `get`, `is` or `has` prefixes for methods with side-effects.	`getUser(): User`	`getUser` *also creates User*	[2, 3]
22	Only use `get` prefix for field accessors that return a value.	`getUser(): User`	`getUser` *fetches an API*	[2, 3]
23	Only use `is` and `has` prefixes for Boolean field accessors.	`isActive(): Boolean`	`isActive(): void`	[2, 3]
24	Only use `set` prefix for field accessors that don't return a value.	`setName(name): void`	`setName(name): String`	[2, 3]
25	Use transformation verbs only for methods that return a transformed value.	`toString(): String`	`toString(): void`	[2, 3]
26	Expecting and getting single instance.	`getUser(): User`	`getUser(): User[]`	[2, 3]
27	Expecting and getting a collection.	`getUsers(): User[]`	`getUsers(): User`	[2, 3]
28	Method name and return type should not contradict.	`getUser(): User`	`getUser(): Database`	[2, 3]

(d) Method Guidelines

Table 7.4: Guidelines used in this chapter to assess the quality of identifiers. Guidelines annotated with [1] have been taken from Butler et al. (2010), [2] from Hilton and Hermans (2017), and [3] from Arnaoudova et al. (2016).

Coding Guidelines

The coding guidelines offer a set of rules for naming identifiers. To construct the dataset, 28 guidelines originally proposed by Hilton and Hermans (2017), Butler et al. (2010) and Arnaoudova et al. (2016) were selected. The guidelines were aggregated by merging similar guidelines and removing overly specific ones, which could rarely be applied to the code samples. The complete list of guidelines used in this chapter is shown in Table 7.4.

The guidelines can be divided into four categories: syntax, vocabulary, data type, and method naming. Nine syntactical guidelines, shown in Table 7.4a, focus on casing, the use of underscores, and the length of identifiers. This can be seen as the most basic level of identifier quality, as many of these guidelines could also be checked by rules without semantic understanding of the code. A more advanced level of identifier quality is covered by the five vocabulary guidelines, shown in Table 7.4b, which include guidelines for the use of dictionary words, the avoidance of abbreviations, and the use of descriptive names. Adhering to these guidelines enables the developer to understand the purpose of the identifier without having to look up its definition. Table 7.4c shows the five data type guidelines, which focus on the consistency of names and types, for example by using singular names for values and plural names for collections. For a developer these guidelines avoid confusion and make the code easier to understand. Finally, the nine method naming guidelines, shown in Table 7.4d, focus on the naming of methods and are only applicable to method names. These guidelines ensure that the method name reflects the method's purpose, and that the method name is consistent with the return type.

Even though some guidelines of lesser difficulty can be detected by rules, such as "*Only use one underscore at a time*", more complex ones require semantic understanding of the functionality of the methods, such as "*Use transformation verbs only for methods that return a transformed value*". Nonetheless, it is interesting to investigate how well an LM adheres to both types of guidelines. This provides an overall impression of how well LMs capture the quality of identifiers.

Anntotation Process

Given the code files and the 28 selected guidelines the annotation process was conducted in two phases by six developers. In the first phase, every identifier in every code files was annotated for their conformance with respect to all guidelines. Out of 2143 identifiers 677 (32%) were found to violate at least one of the guidelines.

Next, in the second phase, the annotators were given a code file, a list of random identifiers from the code file, and a guideline for assessing the quality of the identifiers. They were asked to either correct the identifiers if it did not meet the guideline (shown in green in Figure 7.4c) or to confirm that the identifier was correct and create a violating version if possible (shown in red). When the annotators found the guideline not applicable to the

```java
protected void doXContent(XContentBuilder builder, Params params) throws IOException {
  builder.startObject( ScriptFilterParser.NAME);
  builder.field("script", script);
  if (this.params != null) {
    builder.field("params", this.params);
  }
  if (this.lang != null) {
    builder.field("lang", lang);
  }
  if (filterName != null) {
    builder.field("_name", filterName);
  }
  if (cache != null) {
    builder.field("_cache", cache);
  }
  if (cacheKey != null) {
    builder.field("_cache_key", cacheKey);
  }
  builder.endObject();
}
```

.java

(a) Code

	Guideline
Name	Use dictionary words.
Description	Spell words out in full and define abbreviations for the bounded context.
Violating Example(s)	`acc`, `pos`, `mod`, `auth`, `appCnt`
Details	Only use correctly-spelled dictionary words and abbreviations. Make exceptions for `id` and documented domain-specific language/abbreviations. Spelling mistakes can render names ambiguous, and result in confusing inconsistency. Abbreviations introduce a different kind of ambiguity that the original programmer does not see because they know which word the abbreviation stands for, even if multiple words have that same abbreviation.

(b) Guideline

Identifier	**Violation**	**Correction**
`filterName`	`filName`	
`lang`		`language`
`endObject`		
`builder`	`bldr`	
`startObject`	`sObj`	
`cacheKey`	`cacheK`	
...		

(c) Identifier

Figure 7.4: The annotation process of the manually annotated dataset. The annotators were given the code snippet in 7.4a, the guideline in 7.4b, and a list of random identifiers of the file to annotate, shown in 7.4c. The annotators were asked to annotate at least 5 identifiers either by correcting an identifier that violates the guideline (shown in green in 7.4c) or by creating a violation (red). Thereby, the annotators could also skip identifiers, if they found the guideline not applicable.

```java
public abstract class Filter {

  public abstract boolean shouldRun(Description description);
  public abstract String describe();

  public void apply(Object filterRunner) {
    if (!(filterRunner instanceof Filterable)) {
      return;
    }
    Filterable filterable = (Filterable) filterRunner;
    filterable.filter(this);
  }

  public Filter intersect(final Filter otherAcceptedTests) {
    if (otherAcceptedTests == this || otherAcceptedTests == ALL) {
      return this;
    }
    final Filter theseAcceptedTests = this;
    return new Filter() {
      public boolean shouldRun(Description description) {
        return theseAcceptedTests.shouldRun(description) &&
        ↪  otherAcceptedTests.shouldRun(description);
      }

      public String describe() {
        return theseAcceptedTests.describe() + " and " +
        ↪  otherAcceptedTests.describe();
      }
    };
  }
}
```

.java

(a) Original file without guideline violations.

```java
public abstract class Filter {

  public abstract boolean should_run(Description description);
  public abstract String isDescribed();

  public void apply(Object filterRunner) {
    if (!(filterRunner instanceof Filterable)) {
      return;
    }
    Filterable filterables = (Filterable) filterRunner;
    filterables.filter(this);
  }

  public Filter getIntersection(final Filter otherAcceptedTests){
    if (otherAcceptedTests == this || otherAcceptedTests == ALL) {
      return this;
    }
    final Filter theseAcceptedTests = this;
    return new Filter() {
      public boolean should_run(Description description) {
        return theseAcceptedTests.should_run(description) &&
        ↪  otherAcceptedTests.should_run(description);
      }

      public String isDescribed() {
        return theseAcceptedTests.isDescribed() + " and " +
        ↪  otherAcceptedTests.isDescribed();
      }
    };
  }
}
```

.java

(b) Violated version.

Figure 7.5: Example from our dataset. On the left the (shortened) file without guideline violations, on the right the violated version. We highlighted identifiers by scaling the likelihood-ratio score of the SyntaxPT model linearly between 1–50. Note that even in the original the model finds some identifiers suspicious (left). However, it spots most guideline violations (right) but fails to detect the identifier `filterables` in lines 10–11 (which violates Guideline 13).

identifier, they could skip the identifier. In the example in Figure 7.4c, the guideline is to use dictionary words and no (uncommon) abbreviations. The annotator corrected the (violating) identifier `lang` to `language` and create a violation for the identifier `cacheKey` by shortening it to `cacheK`. In total, 865 randomly selected identifiers were annotated in this way, resulting in 4503 guideline violations and 1700 corrected identifiers[2]. Every annotation from this phase was cross-checked by a second person.

Evaluation Dataset

Once the annotation process was completed, the code files were first normalized to a high-quality state, by replacing all identifiers that violated a guideline with the corrected version, which may not be possible for all identifiers. Next, for each code file 30 random variations were generated: In each file up to 5 identifiers for which a violating version existed were sampled and replaced with the violating version. Note that since some identifiers were left unchanged and violating, they appear in every variation of the file. This is addressed in the evaluation procedure, described in the next section. In total 1770 densely annotated code files were generated, each containing up to 5 identifiers that violate a coding guideline and otherwise only guideline-adhering identifiers.

[2] An identifier can be annotated multiple times for each guideline.

7.5 Experimental Setup

In the preceding sections, multiple strategies to detect guideline violations of code identifiers with LMs were introduced. Additionally, a manually annotated dataset for evaluating these strategies was described. This section presents the experimental setup used to evaluate the effectiveness of the proposed strategies and to compare the performance of the SyntaxPT model developed in Chapter 4 against other state-of-the-art LMs. This section is structured as follows: first, the research questions are outlined; next, the evaluation procedure and metrics used for model comparison are formalized in Section 7.5.2; and finally, the implementation details of the baseline models are provided.

7.5.1 Research Questions

The experiments address the following research questions:

Research Question 7.1: *Which of the proposed strategies is most effective for detecting guideline violations of code identifiers?*

This research question aims to evaluate the effectiveness of the different strategies proposed in this chapter for detecting guideline violations of code identifiers with LMs. To this end, the discriminative approach is compared to the generative approaches. For the generative approaches, different scoring methods, including perplexity, likelihood-ratio, and max-token perplexity score and masking schemes that control the amount of contextual information available to the model are compared. For a fair comparison all these experiments are conducted with the same model, SyntaxPT, which is used as a basis for both the generative and discriminative approaches. The experiments for the generative model use SyntaxPT in a zero-shot setting, while the discriminative approach uses the same model as a basis for fine-tuning.

Research Question 7.2: *How accurately can SyntaxPT spot guideline violations compared to other state-of-the-art LMs?*

To address this research question, the SyntaxPT model from Chapter 4 is compared against other state-of-the-art LMs at the time of publication of the original work. Specifically, the encoder model GraphCodeBert is used as a baseline for the discriminative approach, and the decoder-based InCoder model is used for the generative approach. The rationale behind the selection of these models and their implementation details are provided in Section 7.5.3.

Research Question 7.3: *How accurately can the individual models detect different types of guideline violations?*

The evaluation dataset comprises 28 distinct guidelines that assess the quality of identifiers, which are categorized into four groups: syntax, vocabulary, data type, and method naming,

as shown in Tables 7.4a to 7.4d. This research question aims to determine whether certain guidelines pose more difficult challenges for the models or reveal patterns in their performance. To investigate this, the performance of each model is evaluated on a per-guideline basis.

7.5.2 Evaluation Procedure

With the annotated dataset and a set of models at hand, it is now possible to evaluate how well the models can assess the quality of the identifiers or, in our case, detect violations of the coding guidelines. While this could be evaluated as a classification task using precision and recall—which measure how many of the predicted violations are actual violations and how many of the actual violations are predicted by the model—these measures require a threshold on the models' scores. Moreover, precision and recall do not consider the order or severity of the violations, which is important for practical applications. Rather than using precision and recall, the task is viewed as a *retrieval task*, i.e., to detect and rank the most severe potential violations of the coding guidelines. In a practical scenario, these could then be presented for review to the developer. A retrieval scenario allows us to judge the quality of a model by the distribution of relevant items (here, guideline violations) over its ranked result list, without any thresholding and common retrieval metrics like Mean Average Precision (MAP) can be used for measuring performance.

The standard MAP calculation, as defined in Equation (2.30), is not directly applicable to the evaluation of the models in this context for two reasons. First, the MAP score should be calculated for each guideline independently, as some guidelines have more violating identifiers in the dataset than others. However, different guideline violations can occur simultaneously in an evaluation sample. Second, the dataset contains multiple versions of the same original code file since in each version a random selection of identifiers has been turned violating, while some violating identifiers appear in every version (because they have no non-violating version, see Section 7.4.2). Hence, the MAP calculation is slightly modified, which is explained below.

Here, the 1770 evaluation samples are called *versions*. Each version is a code file in which some identifiers are already violating and some have been replaced with guideline-violating substitutes. Given a version v of a code file c, the models rank all identifiers in v by their likelihood of violating any of the coding guidelines. Each guideline $G \in \mathbb{G}$ is then evaluated independently:

$$\text{MAP} = \frac{1}{|\mathbb{G}|} \sum_{G \in \mathbb{G}} \text{MAP}_{\text{G}}(G) \tag{7.11}$$

The MAP score for guideline G considers only versions of the code files in which an identifier violates G. Additionally, only the identifiers that violate G are considered

relevant in the result list and identifiers that violate other guidelines are filtered. Let us denote the set of all code files in which violations of guideline G occur as $\mathbb{D}_G$. Then the MAP score for guideline G is calculated as:

$$\text{MAP}_\text{G}(G) = \frac{1}{|\mathbb{D}_G|} \sum_{c \in \mathbb{D}_G} \text{AP}(G, c) \tag{7.12}$$

Analogously to the definition of AP in Equation (2.29), the AP is defined based on the precision at k score for each relevant target, in our case violating identifier, where k is the position of the identifier in the ranked list. To account for the different versions of the same code file, the precision at k scores of the identifier is averaged over all versions of the file in which the identifier appears. Formally, let us define the set of all identifiers that violate guideline G in all versions of the file c as $\mathbb{I}_{c,G}$. The average precision for guideline G in file c is then calculated as:

$$\text{AP}(G, c) = \frac{1}{|\mathbb{I}_{c,G}|} \sum_{I \in \mathbb{I}_{c,G}} \text{AvgPrecAtk}(I, c, G) \tag{7.13}$$

This uses the arithmetic mean of the precision at k scores for each identifier I across all versions of the file c in which the guideline G is violated. With the set of all versions of code file c denoted as $\mathbb{S}_{c,G,I}$, the average precision at k score is defined as:

$$\text{AvgPrecAtk}(I, c, G) = \frac{1}{|\mathbb{S}_{c,G,I}|} \sum_{v \in \mathbb{S}_{c,G,I}} \text{PrecAtk}(I, v, G) \tag{7.14}$$

In each version, all identifiers are ranked descending by the score of the model, which indicates the severity of the identifier violating the guideline. When identifier I is ranked at position k in version v, the precision at k is calculated as:

$$\text{PrecAtk}(I, v, G) = \frac{1}{k} \sum_{i=1}^{k} \mathbf{1}_{\text{Identifier at rank } i \text{ violates } G} \tag{7.15}$$

Finally, the overall MAP score is computed by averaging the MAP scores over all guidelines, as detailed in Equation (7.11).

7.5.3 Implementation of Other Language Models

The model is compared to two state-of-the-art LMs pretrained on code: the encoder-based GraphCodeBert (Guo et al. 2021) and the decoder-based InCoder (Fried et al. 2023). This section describes why these models were chosen and how they were adapted for the task of identifying guideline violations in code identifiers.

Discriminative Rating

The discriminative approach employs an encoder architecture, which makes it reasonable to use a state-of-the-art encoder LM as a drop-in replacement for SyntaxPT. This chapter compares SyntaxPT with GraphCodeBert, introduced by Guo et al. (2021). The GraphCodeBert model is an improved version of the CodeBert model. For a comprehensive description of the model, please refer to Section 4.2.

GraphCodeBert is a good candidate for comparison, not only because it is a state-of-the-art encoder model for code understanding tasks, but particularly because it was pretrained with a discriminative learning approach. During pretraining, the model was tasked to discriminate real tokens from replaced ones, along with several other tasks. This is analogous to the approach used in this chapter, albeit with a focus on token-level rather than identifier-level discrimination. In theory, this provides GraphCodeBert with an advantage over the SyntaxPT model, which did not learn to discriminate between real and replaced identifiers during its generative pretraining. SyntaxPT has to learn this task during fine-tuning.

The integration of GraphCodeBert into the approach described in Section 7.3.3 is relatively straightforward. The pretrained GraphCodeBert model replaces the encoder, while the same discriminative fine-tuning approach is kept as in the original model (using the same pooling, classification layers, and training). However, the SyntaxPT model is capable to process code of any length thanks to its bucketed relative positional encoding (see Section 2.3.2). In contrast, GraphCodeBert is limited to a maximum sequence length of 512 tokens, which requires a slight modification of the approach. To process longer code files, the files are split into overlapping chunks of 512 tokens. This ensures the model has sufficient context for each identifier. After processing each chunk separately using the approach described in Section 7.3.3, the scores of identifiers that appear in multiple chunks are averaged before evaluation. A batch size similar to the one used in the reference implementation of GraphCodeBert is used for fine-tuning and a sweep over the learning rate is conducted.

Generative Rating

Good candidate models for the generative approach are models that have been pretrained on identifier deobfuscation. An ideal fit would have been the CodeT5 model proposed by Wang et al. (2021b), because it has been pretrained on identifier deobfuscation. The authors tried to use to released code and model checkpoint of Wang et al. (2021b) for identifier deobfuscation, but the released model appears to lack this functionality. Hence, the model could not be used for comparison. Models trained on identifier deobfuscation are scarce, and the only other model with published checkpoints known to the authors is

the CodeT5 model. An alternative are models that have been pretrained on code infilling tasks, such as the decoder-based InCoder model.

InCoder has been introduced by Fried et al. (2023) as a unified generative model designed for code generation through both left-to-right generation and infilling. Both settings enable the model to perform various code generation tasks. Fried et al. (2023) demonstrated the model's efficacy by evaluating the model on variable renaming in a zero-shot setting. Since this task is similar to variable deobfuscation InCoder is an interesting candidate for comparison.

This chapter uses the released code of the authors of InCoder in its infilling setting to produce input sequences with replaced identifiers. Unlike the method described in Section 7.3.1, where a mask token represents a single identifier and may be used multiple times, InCoder uses unique mask tokens (e.g., `MASK1`, `MASK2`, etc.) for each *occurrence* of an identifier. This requires a slight modification of the approach, since the model predicts the log-likelihood of every occurrence separately. Initial experiments compared min, max, and mean pooling strategies for aggregating the scores of each identifier occurrence, and the mean pooling method yielded the best results. Hence, the mean pooling strategy is used to aggregate the scores of each identifier occurrence before evaluation.

7.5.4 Hardware and Training

A potential thread to validity of the proposed evaluation is that the pretrained LMs could have been trained on the files that were annotated in the manually annotated dataset. This would make the evaluation unfair, because the models' scores for replaced identifiers in the annotated files would likely be higher. To mitigate this for SyntaxPT, not only the repositories used for the evaluation dataset were excluded from the training data of SyntaxPT, but also repositories that contained similar files, such as forks of the original repositories. A simple content-based matching is not sufficient to determine clones, because ongoing development often causes slight variations of file content. However, one can assume that the file-structure changes less frequently, so this is used as a heuristic to exclude similar files. To do so, all repositories that might contain a file from the annotated dataset were collected, by checking whether a file with the same name in the same directory existed in the training data. If this was the case, the repository was excluded from the training data of the generative model.

For the generative models no further training or hyperparameter search is required, since these models are used in a zeroshot fashion. Please refer to Section 4.5 for the training details of the SyntaxPT model. The discriminative approaches were fine-tuned on a single A6000 GPU. A hyperparameter sweep over the learning rate was conducted, and early stopping was used on the validation F1-score of the fine-tuning dataset. Test on the evaluation dataset were conducted only once at the very end for every model.

Discriminative	Perplexity		Likelihood-Ratio		Max-Token Perplexity	
	Mask-Single	Mask-All	Mask-Single	Mask-All	Mask-Single	Mask-All
52.6	55.1	52.8	**62.6**	28.2	56.1	53.2

Table 7.5: A comparison of the identifier quality scoring strategies proposed in this chapter. The table shows the MAP score (%) on the manually annotated dataset, for the experiments that use the SYNTAXPT as the base model. The best performing method is highlighted in bold.

7.6 Results

This section presents the results of the experiments conducted to evaluate the effectiveness of the proposed strategies for detecting guideline violations of code identifiers.

7.6.1 Comparison of Scoring Methods

> *Which of the proposed strategies is most effective for detecting guideline violations of code identifiers?* – RQ 7.1

The first set of experiments aimed to determine the most effective scoring method for identifying guideline violations in code identifiers. Table 7.5 compares the results of the discriminative model and the three generative scoring methods: *Perplexity*, *Likelihood-Ratio*, and *Max-Token Perplexity*, each evaluated with two masking strategies (*Mask-Single* and *Mask-All*) on the manually annotated dataset. Recall that the distinction between the two masking strategies is that Mask-Single masks only the target identifier, while Mask-All masks all identifiers in the code snippet. Allowing the model to see all identifiers in the code snippet may provide additional context that could help the model better assess the quality of the target identifier. However, this may also introduce biases, as the model may be influenced by the quality of the surrounding identifiers. Also, recall that mask-single comes with higher computational costs (see Table 7.5)

The results of the experiments reveal that the Likelihood-Ratio scoring method achieves the highest score of 62.6% MAP, when used with the Mask-Single strategy. With this masking strategy, the Likelihood-Ratio outperforms all other generative and discriminative scoring methods. However, this strategy is also the computationally most expensive scoring function. The Perplexity and Max-Token Perplexity methods perform similarly, with a slight advantage for the Max-Token Perplexity method. Recall that the difference between these two methods was that the Max-Token Perplexity method only considers the perplexity of the most suspicious token and not the entire identifier, as the perplexity method does. This result indicates that it is more relevant whether a model spots subword-level derivations, than rating the full identifier. However, this result strongly depends on the vocabulary used by the model and what it considers a token. For models with large vocabularies both scores should anneal. This result suggests that a model's ability to

identify subword-level derivations is more relevant than its rating of the entire identifier. However, this outcome is highly dependent on the model's vocabulary and its tokenization strategy. For models with large vocabularies, one can expect both scores to converge. The discriminative model achieved a MAP score of 52.6%, which is similar to the Perplexity and Max-Token Perplexity methods, when using the Mask-All strategy.

However, the Likelihood-Ratio method requires multiple forward passes through the encoder-decoder transformer to generate the best version for each identifier before scoring the target identifier in another forward pass. The Mask-Single strategy requires this to be done for each identifier in the code snippet. This makes the Likelihood-Ratio method computationally expensive. The discriminative strategy, on the other hand, requires only a single forward pass through an encoder to score the target identifier, making it the most efficient strategy. For application scenarios in which efficiency is important, the discriminative model may be a suitable choice, as it achieves a competitive performance with only ≈10 p.p. less MAP than the best-performing generative model. It can be hypothesized that the performance of the discriminative model could be further improved by fine-tuning the model with a stronger replacement strategy, or even ground truth sourced from GitHub edits, such as in the work by Chen et al. (2022).

Interestingly, the Mask-Single strategy consistently improved performance across all scoring methods, which supports the importance of contextual information during evaluation. This may be due to the fact that the code files in the evaluation dataset are of high quality and confusions, such as when all identifier names are single characters, are less likely to occur. Investigating the performance of the different strategies on lower-quality code files is left for future work.

7.6.2 Comparison to State-of-the-Art Language Models

> *How accurately can SyntaxPT spot guideline violations compared to other state-of-the-art LMs?* – RQ 7.2

Next, SyntaxPT was compared against two state-of-the-art models, GraphCodeBert and InCoder, in terms of their ability to assess identifier quality. Table 7.6 presents the MAP scores for each model. The results show that SyntaxPT significantly outperforms both GraphCodeBert and InCoder, with a MAP score of 62.6% using the *Likelihood-Ratio (Mask-Single)* method. GraphCodeBert, despite its robust architecture, achieved a lower score of 46.3%. This indicates that the identifier deobfuscation pretraining objective of the base model is more closely aligned with the target task of identifier quality assessment than the token-level discrimination pretraining of GraphCodeBert.

Approach	Model	#Params	Generative Scoring	MAP (%)
Baseline	Random			18.3
Discriminative	GraphCodeBert	125M		46.3
	SyntaxPT$_{\text{discriminative}}$	110M		52.6
Generative	InCoder	1.3B	Perplexity (Mask-Single)	22.7
			Perplexity (Mask-All)	25.3
			Likelihood-Ratio (Mask-Single)	23.4
			Likelihood-Ratio (Mask-All)	23.8
	SyntaxPT	247M	Likelihood-Ratio (Mask-Single)	**62.6**

Table 7.6: Comparison between the models based on SyntaxPT and state-of-the-art models.

InCoder, with its large parameter count, struggled in the generative evaluation, particularly when using the *Mask-All* strategy, suggesting that while it is effective in code generation tasks, it may not be as well-suited for identifier quality assessment without further fine-tuning. As detailed in Section 7.5.3 the InCoder model was pretrained on code infilling tasks, and to score an identifier, new mask tokens need to be introduced for each occurrence of the identifier. When predicting the identifier, the model additionally has to predict whether the occurrences refer to the same identifier or not. SyntaxPT has an advantage in this regard, as a single mask token is used for every occurrence of an identifier, and the model can exploit this bias to its advantage. Another difference lays in the tokenization of the models. InCoder tokenizes code across whitespace and punctuation symbols to represent common code idioms as single tokens. Fried et al. (2023) found that when using the model for predicting single identifier names, breaking tokenization at word boundaries led to a slight decrease in performance. This may have contributed to the model's struggle in the evaluation.

7.6.3 Guideline-specific Analysis

> *Which guideline violations are most challenging for the models to detect?* – RQ 7.3

This research question aims to investigate which guideline violations are most challenging for the models to detect, what factors contribute to these challenges, and how these factors could be addressed to improve model performance. Table 7.7 shows the MAP score on each guideline for the models based on SyntaxPT, as well as both state-of-the-art LMs and a random baseline. Specifically, the best performing generative variant has been the *Likelihood-Ratio (Mask-Single)* method for SyntaxPT, while it has been the *Perplexity (Mask-All)* for InCoder. In Figure 7.6 one can observe a strong variance in the MAP scores across guidelines. The MAP scores for the models based on SyntaxPT range from 30% to 91%, while the random baseline varies from 14% to 25%, and the state-of-the-art models range from 12% to 83%, where the lower end belongs to InCoder and the

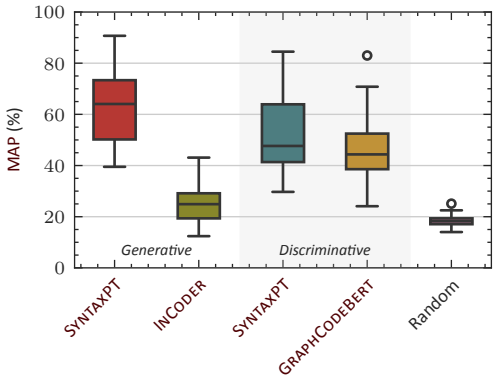


Figure 7.6: Variation of the MAP scores across guideline for the different models.

upper end to GraphCodeBert. The finding that the InCoder model only slightly outperforms the random baseline has been discussed in the previous section. Therefore, the model is excluded from further analysis in this section and in the following, when referring to the generative model, the SyntaxPT model is meant. This strong variation suggests that the models capture guideline information very differently.

First, let us examine whether the models perform differently on guidelines from different categories (syntax, vocabulary, data type, method name). Figure 7.7 shows that the generative SyntaxPT model performs relatively consistently across the four categories, with a slight advantage in method naming guidelines and a slight disadvantage in data type guidelines. Surprisingly, syntax guidelines are not "easier" predicted by the models. So the variation in performance across guidelines is not due to the category of the guideline, but rather the specific guideline itself. In contrast, both discriminative models (SyntaxPT$_{\text{discriminative}}$ and GraphCodeBert) perform notably worse than the generative model on the syntax and vocabulary guidelines, as visible in Figures 7.7a and 7.7b, but at the same time they achieve a similar or even better performance on the data type and method name guidelines compared to the generative model (Figures 7.7c and 7.7d). In a closer inspection in Table 7.7, the discriminative models outperform the generative model in Guidelines 15, 16, and 27 by +19, +7, and +7 p.p. MAP, respectively. These guidelines contain naming conventions for identifier and method names of singular values and collections, such as that collections and methods that return a collection should be named in plural. Additionally, the discriminative models perform better in Guidelines 22 and 24 by +8 and +13 p.p. MAP, respectively, which specify that methods starting with `get` should always return a value, while methods starting with `set` should never return anything.

The better performance of the discriminative models on these guidelines can be attributed to the sampling process with the FastText model, which seems to sample fake identifiers

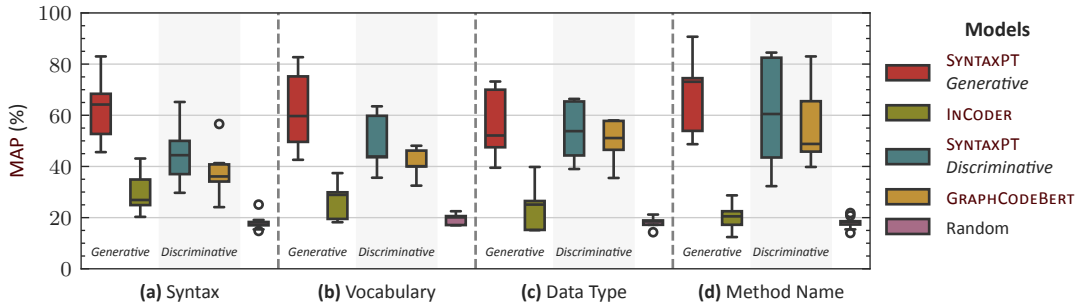


Figure 7.7: Variation of the MAP scores across guideline categories for the different models.

that are effective to detect these types of guideline violations. This hypothesis is supported by a manual inspection of the FastText model, where it could be observed that the model often replaces identifiers with their singular/plural forms. A simple regex check showed that among 306k replacements in the training set of the discriminative model, approximately 1.8% involved replacing a plural form with a singular (3.3k) or a singular with its plural form (2.1k). Furthermore, character n-grams such as `get` and `set` are prevalent in the FastText training data, and the model often samples a similar identifier prefixed with these characters. Obviously, the sampled identifier is then a direct violation of guideline 22 or 24. The frequent occurrence of both types of replacements and their direct relation to the guidelines make it easier for the model to learn these guidelines. This suggests potential for further improving the performance of the discriminative model by exploring other fake identifier sampling approaches that are more closely aligned with the specific guidelines (i.e., a better representation of the counterclass distribution).

As previously explained, many syntax-based guidelines could be detected using simple regular expressions, but most other guidelines require a more fine-grained understanding of the code and semantic reasoning about the context in which the identifier is used. Interestingly, Table 7.7 shows that violations of supposedly easier guidelines, such as those prohibiting double underscores (guideline 4), single-letter abbreviations (guideline 3), or restrictions on the length of identifiers (guidelines 7 and 8), are not detected better than guidelines requiring more complex reasoning. For example, guideline 13 (use a large vocabulary) is detected better than guideline 3 (expand single-letter names) by the generative model, even though the latter is considered easier (75% vs. 46% MAP). This result can be considered promising, as it opens the possibility of combining LM-based scoring with a rule-based approach. Such an approach could be implemented in a post-processing step, where the LM identifies potential guideline violations, which are then checked by a rule-based system. Alternatively, with the previous analysis in mind, which indicated that the fake identifier sampling process of the discriminative model seems to be a good fit for specific guidelines, a rule-based system could be used during

fine-tuning dataset construction to ensure a sufficient number of guideline violations for each guideline. By integrating this into the fake identifier sampling process of the discriminative model the performance on these types of guidelines could be improved. For instance, the discriminative models struggle with guideline 4 (double underscores) and guideline 3 (single-letter abbreviations), possibly due to the FastText model rarely sampling identifiers with these characteristics. To address this, a rule-based system could be employed to generate fake identifiers that more frequently violate these guidelines, which would improve their representation in the training data.

When analyzing the guidelines where the generative model struggles the most, it can be seen that those are guidelines that are likely to be violated frequently in practice. The frequency of guideline violations in the training data is a key factor in the generative model's performance. This observation aligns with the finding that the discriminative model performs well on guidelines frequently violated in its training data, while the inverse holds true for the generative model. The more often a guideline is violated in the pretraining dataset of the generative model, the worse the model performs on that guideline, as reflected in the inverse relationship between perplexity and the model's probability. For example, the model struggles with Guidelines 3 (avoid single-letter abbreviations), 14 (omit type information), and 21 (don't use `get` / `is` / `has` for methods with side effects), which are likely violated frequently in practice. Hence, the generative model performs poorly on these guidelines, as shown in Table 7.7. Again, a more fine-grained and guideline-specific rule-based filtering of the pretraining dataset could help to address this issue.

7.7 Conclusion and Future Work

This chapter presented a novel approach to assess the quality of code identifiers using transformer-based LMs, such as the SyntaxPT model introduced earlier in this thesis. A particularly novel contribution is framing the task as detecting violations of established identifier naming guidelines. This allowed to objectively compare scoring functions for identifiers, including ones that have been proposed by related work (Allamanis et al. 2014). Two primary strategies for scoring identifiers were explored: a generative approach that uses likelihood estimates from the SyntaxPT model, and a discriminative approach that fine-tunes the model to distinguish between original and replaced identifiers.

To evaluate these methods, the first benchmark dedicated to identifier quality assessment was introduced, comprising 6,203 annotations across 28 naming guidelines. Experimental results demonstrated that the generative approach, particularly the Likelihood-Ratio scoring method with the Mask-Single strategy, outperformed the other methods and state-of-the-art LMs such as GraphCodeBert and InCoder. Also, the guideline-specific analysis revealed that the SyntaxPT effectively identifies complex guideline violations that require semantic understanding.

	Guideline	Generative		Discriminative		Random
		Code-Doctor	In-Coder	Code-Doctor	Code-BERT	
1	Apply standard case with rigorous consistency.	**65.9**	20.3	50.0	39.4	17.0
2	Use dictionary words and no (uncommon) abbreviations.	**49.4**	32.7	37.0	34.1	25.1
3	Expand single-letter names (except for control variables).	**45.6**	26.9	30.2	28.9	18.2
4	Only use one underscore at a time.	**83.0**	34.9	45.3	41.3	17.8
5	Only use underscores between words.	**68.4**	24.0	65.2	56.6	18.1
6	Name constant values.	52.7	24.9	**60.8**	40.8	15.4
7	Limit name character length to 20.	**64.2**	43.1	29.7	24.1	19.1
8	Limit name word count to four.	**70.5**	37.7	44.4	36.1	14.8
9	Qualify values with suffixes.	**63.9**	24.9	41.8	34.2	18.3
	Average performance on syntax guidelines	**62.6**	29.9	44.9	37.3	18.2
10	Use a descriptive name that conveys a recognizable concept.	**59.7**	18.2	43.9	40.0	17.1
11	Be precise by identifying specific information and purpose.	**42.6**	19.5	35.6	32.5	20.6
12	Use standard language, avoid humor and ambiguity.	**82.7**	28.9	63.5	48.1	20.1
13	Use a large vocabulary: replace phrases with specific terms.	**75.2**	37.4	59.8	46.2	22.5
14	Don't use prefixes or suffixes that encode the data type.	**49.6**	29.9	43.6	40.1	17.0
	Average performance on vocabulary guidelines	**62.0**	26.8	49.3	41.4	19.5
15	Use singular names for values.	47.5	39.8	**66.4**	58.0	18.9
16	Use plural names for collections.	39.5	15.1	39.0	**46.5**	17.2
17	Use Boolean variable names that imply true or false.	**52.1**	15.2	44.3	35.5	21.2
18	Use positive Boolean names.	**73.2**	26.5	53.8	51.1	14.3
19	Attribute name and type should be consistent.	**70.0**	25.1	65.4	57.8	18.5
	Average performance on data type guidelines	**56.5**	24.3	53.8	49.8	18.0
20	Use a verb-phrase name.	**48.7**	17.2	40.0	42.9	18.6
21	Don't use `get`, `is` or `has` prefixes for methods with side-effects.	**50.4**	20.5	43.5	39.8	17.5
22	Only use `get` prefix for field accessors that return a value.	74.5	18.9	**82.5**	48.8	17.3
23	Only use `is` and `has` prefixes for Boolean field accessors.	**90.7**	28.7	84.5	83.0	20.8
24	Only use `set` prefix for field accessors that don't return a value.	53.9	22.5	60.5	**65.5**	15.4
25	Use transformation verbs only for methods that return a transformed value.	**73.8**	14.3	57.0	45.8	21.8
26	Expecting and getting single instance.	**57.3**	12.4	32.3	47.2	14.0
27	Expecting and getting a collection.	75.8	27.6	**82.6**	70.8	18.3
28	Method name and return type should not contradict.	**73.1**	20.5	69.1	61.9	18.5
	Average performance on method naming guidelines	**66.5**	20.3	61.3	56.2	18.0
	Overall	**62.6**	25.3	52.6	46.3	18.3

Table 7.7: Detailed guideline-specific analysis of the models' performance. The MAP score is reported for each guideline. The best performing model for each guideline is highlighted in bold. The GraphCodeBert model is named CodeBert in the table for brevity.

Future work could explore the integration of rule-based systems for syntax-based guidelines. Such a model could be integrated in the fake identifier sampling process of the discriminative model to improve performance on specific guidelines. Additionally, one could think of creating weakly supervised training data for the discriminative model by using an instruction-based LM to generate fake identifiers that specifically violate certain guidelines. This would allow the discriminative model to learn guidelines more effectively. Since the proposed dataset is build from open-source projects, further evaluation on domain-specific codebases is needed. One could expect the model to find project-specific coding conventions suspicious, which could be addressed by training a local model to reduce false positives. Additionally, since this evaluation is focused on Java, it would be interesting to extend the evaluation to other programming languages and coding conventions (which SyntaxPT supports). Another interesting direction for future work would be to use the model to specifically assess *which* guideline is violated in a given identifier, rather than just providing a score.

The findings of this chapter have implications for both research and practice. In research, the proposed approach can be used to evaluate the quality of identifiers in source code, and the dataset can serve as a benchmark for future research in this area. In practice, the approach can easily be integrated into IDEs as a plugin or code linter. The availability of multiple scoring strategies, offers flexibility in terms of speed and accuracy. For example, developers can select the Likelihood-Ratio scoring for high-accuracy detection or opt for faster but less precise methods depending on the use case. This chapter could also be valuable in educational settings, in teaching students best practices for writing clean, readable code. By integrating this system into programming exercises, educators could provide immediate, actionable feedback to students on their identifier naming choices. Given the probabilistic nature of the model, there will be cases where the feedback is not perfect. This presents an opportunity to foster critical thinking, encouraging students to reflect on both their own coding practices and the output of the model.

8

Conclusion

This thesis has investigated the use of AI models—specifically, transformer-based LMs—to support software engineers with tasks in reuse and quality control. To address these challenges, this thesis investigated several strategies to integrate the structural properties of source code into transformer-based models. Particularly, it was explored how Abstract Syntax Trees (ASTs) can be utilized within self-supervised learning to enhance transformer-based models' code understanding capabilities, learn code retrieval models, and assist developers in practical scenarios.

The thesis began by introducing the Relative Structural Transformer (RST), which demonstrated that adding a structural prior to the transformer through relative positional embeddings and a structural loss function improves performance on code understanding and machine translation tasks when trained without pretraining. The next chapter focussed on improving pretraining for generative transformer models with structural tasks, where the task identifier deobfuscation was extended and the novel task tree-based span selection introduced. The effectiveness of this structural multi-task pretraining strategy was demonstrated with the SYNTAXPT model, which outperformed state-of-the-art models on several CODEXGLUE benchmarks. The third model addressed code reuse by presenting a novel self-supervised strategy to train code retrieval models for Contextualized Code Search (CCS). To this end, AST-based deleaking steps for context-target pairs were introduced, which were shown to reduce overfitting due to leakage patterns and improve performance on CCS. These steps enabled a training without labeled data, resulting in the SYNTAXPT-CCS model. The effectiveness of SYNTAXPT-CCS was validated both in research benchmarks and real-world user studies, for which this thesis presented the CODEBUDDY prototype for practical code search. The studies revealed that even though

SyntaxPT-ccs was trained on open-source data, it could be effectively applied also in proprietary codebases. Finally, the applicability of SyntaxPT to assess the quality of code identifiers using the likelihoods of the LM, with established coding guidelines as a reference, was demonstrated.

8.1 Limitations and Threats to Validity

This chapter provides an overview of the limitations and threats to the validity of the research presented in this thesis. While most details have been addressed in the respective chapters, this section provides a comprehensive overview. One limitation of the user studies on CCS is the relatively small sample size. Also, the controlled setting in Study A in Section 6.4 may not fully express the regular work activities of developers. Even though the findings indicate that developers found CodeBuddy useful, larger-scale studies are necessary to generalize these results, and to identify potential issues that may arise in real-world scenarios. Study B in Section 6.5 was a step in this direction, but more extensive experiments with a larger number of participants are needed to validate the effectiveness of CodeBuddy in practical software development scenarios.

Data quality is another concern—as often in ML experiments. The evaluation datasets presented in this thesis were carefully curated, with dense annotations and manual verification. In the case of the identifier quality assessment dataset, there is a potential risk that an annotator's individual perception of identifier quality may have influenced the annotation process, even though guidelines, examples, and detailed instructions were provided. To circumvent this threat, the annotations were reviewed by another annotator.

Also, the quality of the training data used for the models is less certain, as it was sourced from open-source repositories, and may contain errors, outdated elements, or code that does not adhere to best practices. This may not be a significant issue for code retrieval, because one can ensure the indexed codebase remains up-to-date and high-quality. However, for tasks like identifier quality assessment, the quality of the training data is crucial. Surely, there is a discrepancy between coding guidelines and actual developer behavior. Developers may intentionally deviate from guidelines for various reasons, such as project-specific requirements or personal preferences. This factor potentially affects the model's effectiveness in estimating identifier quality.

Another threat to validity is the possibility of target data leakage during training. Because of the large amount of open-source data used for training the SyntaxPT and SyntaxPT-ccs models, there is a risk that the models may have seen evaluation data during training. This is a key issue with all LLMs, as they are known to memorize examples from the training data. This is now discussed in more detail for each chapter.

- The experiments in Chapter 3 are unlikely to be affected by this issue, as the Rst model was trained without pretraining, and the datasets were split by project to avoid such leakage.
- For the experiments in Chapter 4 the pretraining dataset may include parts of CodeXglue. However, the benefit of structural pretraining were not solely determined by a comparison to the state-of-the-art, but more importantly to non-structural baselines trained on the same data.
- Regarding Chapter 5, there is a potential risk, that data from Cocos may have been included in the pretraining data, which could have influenced the model's performance. Although the same training data was used for the ablation studies on the Cocos dataset, there remains a slight chance that the model may have learned to associate certain contexts with specific targets during generative pretraining. This—although unlikely—could have impacted the comparison with bm25.
- Finally, for the identifier quality assessment in Chapter 7, data leakage is a serious problem, since the model would spot manually inserted violations easily if it had memorized the original code file. Therefore, the author of this thesis took great care to mitigate this risk by excluding data from the same filenames and directories during pretraining. However, this is a heuristic and may not be foolproof, and some degree of leakage may still be unavoidable due to similar patterns across projects.

8.2 Future Work

While this thesis has demonstrated the effectiveness of integrating structural information into transformer models, there remains potential for further exploration. One promising direction of future work is the scaling of model sizes: Larger models, as evidenced by the ubiquity of LLMs since late 2022, have shown remarkable capabilities in code understanding and generation. Whether the benefit of structural pretraining tasks extends to larger models remains an open question, which this thesis could not address due to computational constraints.

Also, relying solely on code generation LLMs is not without drawbacks. Generative transformers, come with increased computational costs and may produce less reliable outputs. As reported by GitClear et al. (2024), with the increased used of code generation LLMs there has been "a significant uptick in churn code [the percentage of lines that are reverted or updated less than two weeks after being authored], and a concerning decrease in code reuse". Hence, tools that enable opportunistic code reuse, such as the proposed CodeBuddy prototype for CCS, remain highly relevant, even in the era of LLMs.

Therefore, it would be interesting to investigate if CodeBuddy could be scaled in four areas: scaling up the SyntaxPT-ccs model, increasing the amount of negatives samples

in the contrastive loss, extending the context window, and exploring larger codebases. Scaling up the SyntaxPT-ccs model and the amount of negatives samples in the contrastive loss could potentially improve their retrieval performance substantially. This potential improvement is supported by the experiment with OpenAI's code embeddings (OpenAI 2024b) in Section 5.6.4, which revealed that larger models can provide strong retrieval improvements, even though the presumably much smaller SyntaxPT-ccs model was already competitive.

Another area for future work is extending the contextual information provided to the models. All experiments in this thesis were conducted with a relatively limited view of the user's coding context, focusing on a single file (CodeBuddy in Chapter 6) or even a single method (in Chapters 3 to 5 and 7). Current transformer architectures are limited in the length of input sequences they can process, which constrains their ability to consider broader project contexts. Project structure, other files, or even the available API, along with the API's documentation are often equally important for understanding and reusing code. Exploring strategies to integrate this relevant information into the models could lead to more comprehensive code understanding and more accurate retrieval capabilities. Even though recent larger models with larger context windows, using linear attention (Kitaev et al. 2020) or state space model architectures (Ren et al. 2024) may alleviate this limitation to some extent[1], context windows are still limited and strategies to detect and incorporate relevant information from project contexts are necessary.

One could also explore how well the SyntaxPT-ccs model performs on larger codebases. In the experiments in this thesis the codebase was medium-sized with ranging from 5,000 to 15,000 files. Web-scale codebases could potentially introduce new challenges. It would be interesting to see how the code retrieval model performs on such large codebases.

Furthermore, the work in Chapter 7 on identifier quality touches just a small part of the code quality spectrum, and similar strategies could be applied to many other aspects of code quality. One could potentially utilize the token likelihoods from LMs or even LLMs to detect code smells, bugs, design patterns, and antipatterns. This could be done all based on the assumptions that developers are more likely to write code that adheres to best practices and that "bad" code is less likely (to be present in LLMs training sets, and thus less likely to be produced by the LLM). However, this would presumably require more sophisticated strategies than the perplexity threshold used in Chapter 7 in order to spot patterns or bugs. Also, to support inexperienced developers in understanding their code, one could not only pinpoint areas in the code which the LLM estimates as unlikely, but also subsequently prompt an instruction-based LLM to provide explanations or suggestions for improvement. An IDE plugin could then present these suggestions to the developer in a non-intrusive way, similar to code linting tools.

[1] For instance, the Google Gemini model has a 2 million token context window (Reid et al. 2024).

Part III

Appendix

A

Appendix to Part I

A.1 Relative Structural Transformer

A.1.1 Hyperparameters and Datasets

	java-small		java-med		java-large		CodeSearchNet		FunCom		IWSLT'14	
	Baseline	**RST**	**Baseline**	**RST**	**Baseline**	**RST**	**Baseline**	**RST**	**Baseline**	**RST**	**Baseline**	**RST**
Warmup Updates	4000	4000	4000	4000	10000	10000	4000	4000	4000	4000	4000	4000
Max Epoch	30	30	60	60	60	60	60	60	60	60	70	70
Validation Metric	F	F	F	F	F	F	BLEU	BLEU	BLEU	BLEU	BLEU	BLEU
Validation Performance	44.84	46.19	56.05	58.61	63.31	63.66	8.32*	8.92*	23.42	23.91	29.69	30.07
Max Source Positions	512	512	512	512	512	512	1024	1024	512	512	1024	1024
Max Target Positions	80	80	80	80	80	80	1024	1024	30	30	1024	1024
Batch Size (in tokens/batch/gpu)	8192	8192	8192	8192	8192	8192	6146	6146	6146	6146	4096	10240
Accumulate Gradients (in batches)	8	8	8	8	8	8	6	6	6	6	-	-
Share Embeddings (between Encoder/Decoder)	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	No	No
Relationship Type	-	Movements	-	Movements	-	Path-Length	-	Movements	-	Movements	-	Path-Length
k	-	2	-	2	-	8	-	2	-	2	-	8
γ_{lca}	-	0.3	-	0.3	-	0.3	-	0.3	-	0.3	-	0.3
Parameters (Million)	38.76	38.79	39.3	39.9	47.5	47.6	39.8	40.4	47.5	48.3	39.5	40
Average Runtime (hours)	10	12	22	25	66	71	20	23	16	19	7	9

Table A.1: Additional hyperparameters for the experiments in Tables 3.1 and 3.4. (*) denotes that a different BLEU implementation was used during validation than for testing. For all experiments *Label Smoothing*=0.1, *Learning Rate*=5e-4, *Optimizer*: Adam, *Adam-Betas*=0.9, 0.98 and *Weight Decay*=0.0001. Previously published in **Villmow** et al. (2021b)

	java-small	java-med	java-large	CodeSearchNet	FunCom	IWSLT'14
Samples Train	665,115	3,004,536	15,344,512	908,224	1,937,136	160,239
Samples Valid	23,505	410,699	320,866	44,689	106,153	7,283
Samples Test	56,165	411,751	417,003	52,561	52,561	6,750

Table A.2: Statistics of the datasets used in the experiments for Chapter 3. Previously published in **Villmow** et al. (2021b) ©2021 IEEE.

A.1.2 Sample Predictions

```java
public Throwable blockingGetError() {
    if (getCount() != 0) {
            try {
                BlockingHelper.verifyNonBlocking();
                await();
            } catch(InterruptedException ex) {
                dispose();
                return ex;
            }
        }
    return error;
}
```

Model	Prediction
Target	Block until the latch is counted down and return the error received or null if no error happened.
Transformer	Returns an error if there is one. Otherwise returns nil.
RST	This method blocks until there is an error or the end of the queue is reached.

(a)

```java
public static ScheduledExecutorService create(ThreadFactory factory){
    final ScheduledExecutorService exec = Executors.newScheduledThreadPool(1, factory);
    tryPutIntoPool(PURGE_ENABLED, exec);
    return exec;
}
```

Model	Prediction
Target	Creates a ScheduledExecutorService with the given factory.
Transformer	Create a new ScheduledExecutorService.
RST	Creates a ScheduledExecutorService with the given ThreadFactory.

(b)

Figure A.1: Predictions with RST for code summarization on CODESEARCHNET.

A.2 Structural Transformer

A.2.1 Datasets

```sql
SELECT lang, COUNT(*) AS repos,
  ARRAY_AGG(STRUCT(name, stars, branch) ORDER BY stars DESC LIMIT 50000) AS repo
FROM (
  SELECT repo.name,
    MAX(CAST(JSON_EXTRACT_SCALAR(payload,
    '$.pull_request.base.repo.stargazers_count') AS INT64)) AS stars,
    JSON_EXTRACT_SCALAR(payload,
    '$.pull_request.base.repo.language') AS lang,
    JSON_EXTRACT_SCALAR(payload,
    '$.pull_request.base.repo.default_branch') AS branch
  FROM `githubarchive.month.202108`
  WHERE type = 'PullRequestEvent'
  GROUP BY repo.name, branch, lang
)
WHERE stars > 10
GROUP BY lang
ORDER BY repos DESC;
```

.sql

Figure A.2: BigQuery SQL query to obtain repositories with more than 10 stars and active pull requests in August 2021. A separate query is executed for every month between April and September 2021 and the results are combined.

A.2.2 TensorTree Library

```python
# Encode the Fibonacci function into a TensorTree object.
code = '''def fib(n):
    if n <= 1: return n
    return fib(n-1) + fib(n-2)'''
tree: TensorTree = tokenizer.encode_to_tree(code, "python")

# Access and print various structural details of the tree.
print("Node data (pre-order sequence of nodes):", tree.node_data)
# Output: tensor([31314, 31298,   402,    14, 31383,     5, ...])
print("Parent indices of each node:", tree.parents)
# Output: tensor([0,  0,  1,  1,  1,  4, ...])
print("Number of descendants for each node:", tree.descendants)
# Output: tensor([76, 75,  0,  0,  3,  2,  ...])

# Retrieve and decode the leaf nodes, which are the tokens with no children.
leaves = tree.node_data[tree.descendants == 0]
decoded_leaves = tokenizer.decode(leaves)
print("Decoded leaf nodes back to code:\n", decoded_leaves)
# Output:
# def fib(n):
#     if n <= 1: return n
#     return fib(n-1) + fib(n-2)

# Example of modifying the tree by masking identifiers.
mask_ids = get_mask_token_ids()[1:]  # e.g.\ [30000, 30001, ...]
identifier_nodes = (tree.node_data == 31383).nonzero()  # Id for [identifier]
masked_tree = tree.delete_nodes(
    identifier_nodes.squeeze(), replacements=mask_ids[:len(identifier_nodes)]
)

# View the code with masked identifiers.
masked_code = tokenizer.decode(masked_tree.leaves())
print("Code with masked identifiers:\n", masked_code)
# Output:
# def MASK0(MASK1):
#     if MASK2 <= 1: return MASK3
#     return MASK4(MASK5-1) + MASK6(MASK7-2)
```
.py

Figure A.3: Example of a common workflow using the TENSORTREE library implementing parts of the identifier deobfuscation task (Section 4.4.1). First, the Fibonacci function is encoded into a `TensorTree` object. The leaves of this tree can be easily selected and are used as input to the model. The tree can be decoded back to the original code. To find the indices of identifier nodes a mask over the nodes in the tree is computed. Given the identifier indices, a bulk operation can be used to replace every identifier subtree in a single operation with mask tokens and return a new tree with those identifiers replaced.

```python
# pretty print the tree of token IDs
masked_tree.pprint()
# TensorTree():
# 0. 31314
# ├── 1. 31298
#     ├── 2. 402
#     ├── 3. 14
#     ├── 4. 31414
#     ├── 5. 31280
#     |   ├── 6. 22
#     |   ├── 7. 31415
#     |   ├── 8. 23
#     ├── 9. [...]

# convert the tree of IDs back to a tree of strings
tokenizer.decode_tree(masked_tree).pprint()
# TensorTree():
#  0. [module]
# ├── 1. [function_definition]
#     ├── 2. def
#     ├── 3. ·
#     ├── 4. MASK0
#     ├── 5. [parameters]
#     |   ├── 6. (
#     |   ├── 7. MASK1
#     |   ├── 8. )
#     ├── 9. :
#     ├── 10. [BPE]
#     |   ├── 11. \n
#     |   ├── 12. ····
#     ├── 13. [block]
#         ├── 14. [if_statement]
#         |   ├── 15. if
#         |   ├── 16. ·
#         |   ├── 17. [comparison_operator]
#         |   |   ├── 18. MASK2
#         |   |   ├── 19. ·
#         |   |   ├── 20. <=
#         |   |   ├── 21. ·
#         |   |   ├── 22. [integer]
#         |   |       ├── 23. 1
#         |   ├── 24. :
#         |   ├── 25. ·
#         |   ├── 26. [block]
#         |       ├── 27. [return_statement]
#         |           ├── 28. return
#         |           ├── 29. ·
#         |           ├── 30. MASK3
#         ├── 31. [BPE]
#         |   ├── 32. \n
#         |   ├── 33. ····
#         ├── 34. [return_statement]
```

.py

```python
# pretty print the tree of strings (before masking)
tokenizer.decode_tree(tree).pprint()
# TensorTree():
#  0. [module]
# ├── 1. [function_definition]
#     ├── 2. def
#     ├── 3. ·
#     ├── 4. [identifier]
#     |   ├── 5. [BPE]
#     |       ├── 6. fi
#     |       ├── 7. b
#     ├── 8. [parameters]
#     |   ├── 9. (
#     |   ├── 10. [identifier]
#     |   |   ├── 11. n
#     |   ├── 12. )
#     ├── 13. :
#     ├── 14. [BPE]
#     |   ├── 15. \n
#     |   ├── 16. ····
#     ├── 17. [block]
#         ├── 18. [if_statement]
#         |   ├── 19. if
#         |   ├── 20. ·
#         |   ├── 21. [comparison_operator]
#         |   |   ├── 22. [identifier]
#         |   |   |   ├── 23. n
#         |   |   ├── 24. ·
#         |   |   ├── 25. <=
#         |   |   ├── 26. ·
#         |   |   ├── 27. [integer]
#         |   |       ├── 28. 1
#         |   ├── 29. :
#         |   ├── 30. ·
#         |   ├── 31. [block]
#         |       ├── 32. [return_statement]
#         |           ├── 33. return
#         |           ├── 34. ·
#         |           ├── 35. [identifier]
#         |               ├── 36. n
#         ├── 37. [BPE]
#         |   ├── 38. \n
#         |   ├── 39. ····
#         ├── 40. [return_statement]
#             ├── 41. return
#             ├── 42. ·
#             ├── 43. [binary_operator]
#                 ├── 44. [call]
#                 |   ├── 45. [identifier]
#                 |   |   ├── 46. [BPE]
#                 |   |       ├── 47. fi
#                 |   |       ├── 48. b
```

.py

Figure A.4: Content of the two trees `tree` and `masked_tree` from Figure A.3, decoded back to trees of strings and printed with the TENSORTREE library.

A.3 COCOS Examples

```java
public boolean extract(File f, String folder){
  Enumeration entries;
  ZipFile zipFile;
  try {
    zipFile = new ZipFile(f);
    entries = zipFile.getEntries();
    <CURSOR
    zipFile.close();
  } catch (IOException ioe) {
    this.errMsg = ioe.getMessage();
    Malgn.errorLog(
      "{Zip.unzip} " + ioe.getMessage()
    );
    return false;
  }
  return true;
}
```

(a) Incomplete and masked query x from the COCOS dataset. Code that extracts elements from a zip file needs to be found.

```java
while (entries.hasMoreElements()) {
  ZipArchiveEntry entry = (ZipArchiveEntry)
  ↪ entries.nextElement();
  if (entry == null) continue;
  String path = folder + "/" +
  ↪ entry.getName().replace('\\', '/');
  if (!entry.isDirectory()) {
    File destFile = new File(path);
    String parent = destFile.getParent();
    if (parent != null) {
      File parentFile = new File(parent);
      if (!parentFile.exists()) {
        parentFile.mkdirs();
      }
    }
    copyInputStream(
      zipFile.getInputStream(entry),
      new BufferedOutputStream(
        new FileOutputStream(destFile)
      )
    );
  }
}
```

(b) The masked section y manually selected from c (a). It has been re-formatted for better readability. This is neither part of the query nor used when evaluating (a).

```java
ArchiveEntry ae = zis.getNextEntry();
while(ae != null) {
  // Resolve new file
  File newFile = new File(outputdir + File.separator + ae.getName());
  // Create parent directories if not exists
  if(!newFile.getParentFile().exists())
    newFile.getParentFile().mkdirs();
  if(ae.isDirectory()) {
    // create if not exists
    if(!newFile.exists()) newFile.mkdir();
  } else { // If file, write file
    FileOutputStream fos = new FileOutputStream(newFile);
    int len;
    while((len = zis.read(buffer)) > 0) {
      fos.write(buffer, 0, len);
    }
    fos.close();
  }
  // Proceed to the next entry in the zip file
  ae = zis.getNextEntry();
}
```

(c) Possible solution, that implements the same functionality as the target in (b).

Figure A.5: Example of the COCOS dataset.

B

Appendix to Part II

B.1 Evaluating Contextualized Code Search in Practical User Studies

This is the Appendix to Chapter 6.

B.1.1 Indexing Strategy

The following nonterminals in Table B.1 have been used in CodeBuddy to discover candidate snippets for indexing. Note that these nonterminals have been used as starting points to build n-grams. For example, when an `ExpressionStatement` node (that is part of the list) neighbors a `VariableDeclaration` node (not part) in the AST, their n-gram is indexed as well. The full algorithm is described in Section 6.3.3.

Nonterminal Type	Nonterminal Type	Nonterminal Type
`IfStatement`	`StatementBlock`	`ClassFunction`
`ForStatement`	`ExpressionStatement`	`Function`
`FunctionDeclaration`	`Program`	`ForClause`
`MethodDeclaration`	`Module`	`IfExpression`
`MethodDefinition`	`ClassBody`	`TryExpression`
`Method`	`ClassDeclaration`	`Class`
`BlockArgument`	`ClassDefinition`	`Annotation`
`Block`	`FunctionDefinition`	`MarkerAnnotation`

Table B.1: List of nonterminal types used in CodeBuddy to detect possible targets for indexing.

B.1.2 Example Solutions in Study A

The following two figures show solutions for the original, unaltered exercise of the task shown in Figure 6.7 that have been part of the search index.

```java
import java.util.LinkedList;
import java.util.List;

public class Sieve {
    static boolean[] sieve(int n) {
        boolean[] result = new boolean[n + 1];
        for (int i = 0; i <= n; i++)
            result[i] = true;

        for (int p = 2; p * p <= n; p++) {
            // If prime[p] is not changed, then it
            ↪ is a prime
            if (result[p] == true) {
                // Update all multiples of p
                for (int i = p * p; i <= n; i += p)
                    result[i] = false;
            }
        }
        return result;
    }

    static List<Integer> primesUpTo(int n) {
        List<Integer> result = new LinkedList<>();
        boolean[] primeNumber = sieve(n);
        for (int i = 2; i <= n; i++) {
            if (primeNumber[i])
                result.add(i);
        }
        return result;
    }

    static int[] primeFactors(int n) {
        int[] result = new int[n];
        int sqrt = (int) Math.sqrt(n);
        for (int div = 2; div * div <= n; div++) {
            while (n % div == 0) {
                n = n / div;
                System.out.println(div + " ");

            }
        }
        if (n != 1) {
            System.out.println(n);
        }
        return result;
    }
}
```

.java

Figure B.1: A solution to the original exercise and relevant for the task shown in Figure 6.7.

```java
import java.util.ArrayList;

public class Sieve {
  private static final int n = 50;
  private static boolean[] pr = new boolean[n];

  static boolean[] sieve(int n) {
    for (int k = 2; k <= n; k++) {
      boolean primzahl = true;
      for (int i = 2; i <= k / 2; i++) {
        if (k % i == 0) {
          primzahl = false;
          System.out.println(k + " ist keine Primzahl");
          break;
        }
      }
      if (primzahl == true) {
        System.out.println(k + " ist eine Primzahl");
      }
    }
    boolean[] result = new boolean[n + 1];
    return result;
  }

  private static int[] list() { // Die Methode erzeugt ein int-Array, das die die Zahlen von 2 bis zur Obergrenze speichert.
    int[] list = new int[n]; // Beim Durchlauf werden die Werte darin abgelegt und beim jeweiligen Index der zugehörige boolsche Wert in das Hilfsarray pr eingetragen.
    for (int i = 2; i <= list.length; ++i) { // die 2 und alle ungeraden Zahlen als Primzahlen mit true, alle anderen bereis mit false markiert, da gerade Zahlen als Vielfache von 2 keine Prinzahlen sein können.
      list[i - 2] = i;
      pr[i - 2] = i == 2 || i % 2 == 1 ? true : false;  //Der Fragezeichen-Operator ist eine Kurzversion der If-Else-Anweisung.
    }
    return list;
  }
```

.java

```java
  static ArrayList<Integer> primesUpTo(int[] o) { // Ihr wird das Array mit den Werten, das von list() zurückgegeben wird, als Parameter übergeben.
    ArrayList<Integer> result = new ArrayList<Integer>(); //Im Methodenkörper wird zunächst eine leere ArrayList erzeugt, die später alle Primzahlen aufnimmt und in der Schleife werden die Zahlend von 2 bis 50 geprüft.
    for (int i = 2; i <= n; ++i) {
      if (pr[i - 2]) { //Ist der jeweilige Wert des Zahlenarrays dort mit true als Primzahl gekennzeichnet, so wird er in die ArrayList eingetragen.
        result.add(o[i - 2]);
        for (int j = i * i; j <= n + 1; j += i) {
          pr[j - 2] = false;
        }
      }
    }
    return result;
  }

  static int[] primeFactors(int n) {
    System.out.println("------");
    System.out.println("Teilfaktoren:");
    for (int i = 2; i <= n; i++) {
      while (n % i == 0) {
        n = n / i;
        System.out.println(i);
      }
    }
    int[] result = null;
    return result;
  }

  private static void ausgeben(ArrayList<Integer> result) {
    for (int i : result) {
      System.out.println(i);
    }
  }

  public static void main(String[] args) {
    sieve(50);
    ausgeben(primesUpTo(list()));
    primeFactors(50);
  }
}
```

.java

Figure B.2: A solution to the original exercise and relevant for the task shown in Figure 6.7.

B.1.3 Example Retrieval in Study A

An example of a real search request in Study A is shown in Figure B.3.

```java
public class Primes {
  static boolean[] filter(int n) {
    boolean[] array = new boolean[n + 1];
    for (int i = 0; i < array.length; i++) array[i]
    ↪  = true;
    for (int j = 2; j < array.length; j++) {
      |←CURSOR
    }
    return array;
  }
}
```

.java

(a) Search Request

```java
for (int i = 2; i <= n; ++i) {
  if (pr[i - 2]) { //Ist der jeweilige Wert des
  ↪  Zahlenarrays dort mit true als Primzahl
  ↪  gekennzeichnet, so wird er in die ArrayList
  ↪  eingetragen.
    result.add(o[i - 2]);
    for (int j = i * i; j <= n + 1; j += i){
      pr[j - 2] = false;
    }
  }
}
return result;
```

.java

(b) Result 1: `103/02_Sieve.java`

```java
    result[1] = false;
    result[2] = true;
    for (int i = 2; i < result.length; i++) {
      for (int j = i + 1; j < result.length; j++) {
        if (j % i == 0) {
          result[j] = false;
        }
      }
    }


```

.java

(c) Result 2: `105/09_Sieve.java` ★★★★★

```java
  boolean[] ergebnis = new boolean[N + 1];
  for (int i = 0; i <= N; i++) {
    ergebnis[i] = true;
    for (int j = i - 1; j >= 2; j--) {
      if (i % j == 0) {
        ergebnis[i] = false;
      }
    }
  }
  return ergebnis;
}
```

.java

(d) Result 3: `007/07_Foo.java`

Figure B.3: On the top left, a search request for the prime number filter exercise shown in Figure 6.7. The remaining sub-figures show the top three search results retrieved by CodeBuddy, where the second result was rated as highly relevant by a participant.

9 Glossary

9.1 Abbreviations

AI	Artificial Intelligence. *Pages:* xiii, 1, 10, 49, 52, 141, 149, 186, 207
API	Application Programming Interface. *Pages:* 126–128, 152, 210
AST	Abstract Syntax Tree. *Pages:* iii, v, 3–5, 11, 15–18, 50, 54, 60, 63, 70, 75, 76, 78–80, 83, 84, 117, 125, 126, 145, 154, 156, 171, 207, 217
BPE	Byte Pair Encoding. *Pages:* 14, 53, 63, 65–68, 81, 94–96, 179
CBOW	Continuous-Bag-Of-Words. *Pages:* 37, 38, 187
CCS	Contextualized Code Search. *Pages:* iii–vi, 6–8, 24, 44, 45, 120–125, 127, 128, 133, 135, 137, 139–141, 145–150, 153, 155, 157, 161–166, 171, 207–209
CNN	Convolutional Neural Network. *Pages:* 52
CST	Concrete Syntax Tree. *Pages:* 11, 16, 18, 95–97
CT	Cloze Task. *Pages:* 125, 128, 135, 137, 141
DE	dedenting. *Pages:* 129, 132, 136, 141, 154
GEGLU	Gated Gaussian Error Linear Unit. *Pages:* 94
GELU	Gaussian Error Linear Unit. *Pages:* 33
GPU	Graphics Processing Unit. *Pages:* 52, 59, 99, 100, 103, 104, 136, 198
IDE	Integrated Development Environment. *Pages:* 91, 120, 121, 173, 177, 205, 210
IDF	Inverse Document Frequency. *Pages:* 42
IM	mutual identifier masking. *Pages:* 127, 129, 131, 133, 136, 141, 154
IoU	Intersection over Union. *Pages:* 158

IR	Information Retrieval. *Pages:* 25, 27–29, 31, 39, 40, 42–44, 125, 138, 142, 153
IT	Information Technology. *Pages:* 1
LCA	Lowest Common Ancestor. *Pages:* iii, vi, 5, 17, 51, 55–57, 59–61, 66, 70–72
LLM	Large Language Model. *Pages:* 3, 41, 208–210
LM	Language Model. *Pages:* iii, iv, vi, 5, 6, 8, 9, 37, 39, 41, 44, 45, 76–81, 85, 90, 92, 93, 95, 99, 100, 104, 116, 117, 119, 120, 126, 145, 154, 163, 166, 167, 171–175, 178–182, 185, 191, 193, 194, 196, 197, 199, 200, 202, 203, 205, 207, 208, 210, 231
LOC	Lines of Code. *Pages:* 171, 189
LSTM	Long-Short-Term-Memory Network. *Pages:* 39, 52, 54, 76, 81, 82
ML	Machine Learning. *Pages:* iii, v, 2–5, 7, 18, 21, 22, 208
MLM	Masked Language Modeling. *Pages:* 39, 76–78, 81–85, 87, 90, 117, 122
NLP	Natural Language Processing. *Pages:* iii, v, 2, 3, 5, 6, 11, 13, 15, 37, 39, 44, 50–53, 59, 63, 75–77, 80–82, 85, 122, 125, 126, 142, 157, 172, 185, 187
NMS	Non-Maximum Suppression. *Pages:* 157
OOV	out-of-vocabulary. *Pages:* 13, 14, 53
p.p.	percentage points. *Pages:* 51, 67, 68, 70, 71, 73, 107, 110–112, 115, 116, 125, 139, 140, 199, 201
RAG	Retrieval-Augmented Generation. *Pages:* 3, 140, 142, 163, 167
ReLU	Rectified Linear Unit. *Pages:* xv, 33, 59, 60, 185, 186
REST	Representational State Transfer. *Terms:* RESTful API. *Pages:* 152
RNN	Recurrent Neural Network. *Pages:* 34, 52, 53, 82
TS	tree-based span selection. *Pages:* 129, 130, 132, 133, 135, 136, 141, 154
t-SNE	t-distributed Stochastic Neighbor Embedding. *Pages:* 137
UI	User Interface. *Pages:* 147
UX	User Experience. *Pages:* 151

9.2 Datasets

BigCloneBench — A clone detection dataset introduced by Svajlenko and Roy (2015). *Pages:* 133

Cocos — COntextualized COde Search Dataset. *Contextualized Code Search* dataset introduced in Chapter 5 and by **Villmow** et al. (2022). *Pages:* iii, vi, 7, 124, 125, 133–135, 137, 140, 141, 167, 209

CodeSearchNet — *Code summarization* dataset in six programming languages introduced by Husain et al. (2019). *Pages:* 64, 68, 82, 90, 92, 93, 101, 104, 110, 115, 134

CodeXglue — A code understanding benchmark that spans over various tasks, introduced by Lu et al. (2021). *Pages:* xiii, 6, 20, 80, 82, 83, 92, 93, 100, 102–106, 109, 111, 118, 124, 125, 135, 136, 139, 141, 145, 207, 209

Devign — A defect detection dataset, which is part of the CodeXglue benchmark, introduced by Lu et al. (2021). *Pages:* 101

FunCom — *Code summarization* dataset for Java functions introduced by LeClair and McMillan (2019). *Pages:* 64, 67

Iwslt'14 — A *machine translation* dataset introduced by Cettolo et al. (2014). *Pages:* 65

Java-large — Large version of a *method naming* dataset introduced by Alon et al. (2019a) (10000 repositories). *Pages:* 64, 66

Java-med — Medium version of a *method naming* dataset introduced by Alon et al. (2019a) (1200 repositories). *Pages:* 64, 66, 69

Java-small — Small version of a *method naming* dataset introduced by Alon et al. (2019a) (100 repositories). *Pages:* 64, 66

Poj-104 — *Clone detection* dataset, which is part of the CodeXglue benchmark. *Pages:* 101, 105, 116

9.3 Metrics

accuracy
: Proportion of examples for which the model predicted the correct output (Goodfellow et al. 2016, p. 103). *Definition:* Equation (2.23). *Pages:* 27, 28, 102–105, 107, 109, 110, 112, 125, 139, 140

AP
: Average Precision. IR metric that measures the area under the precision-recall curve. *Definition:* Equation (2.29). *Pages:* 30, 195

BLEU
: Bilingual Evaluation Understudy. Measures similarity between a machine-generated sequence and one or more human reference sequences by computing the n-gram overlap between them. *Definition:* Equation (2.25). *Pages:* xiii, 27–29, 36, 64, 65, 68, 102, 104, 107, 109–111, 115

EM
: Exact Match. Exact match metric that measures the fraction of examples for which the model predicted the correct sequence. Synonym for accuracy. *Pages:* 102, 107, 109–112

F1-score
: The F1-score is the harmonic mean of precision and recall. *Definition:* Equation (2.22). *Pages:* 27, 28, 51, 64, 66, 67, 70, 71, 73, 198

MAP
: Mean Average Precision. IR metric that measures the average of the AP over multiple queries. *Definition:* Equation (2.30). *Pages:* 27, 30, 105–107, 109, 110, 115, 116, 125, 134, 136–141, 194–196, 198–202

MRR
: Mean Reciprocal Rank. IR metric that measures the average of the reciprocal ranks of the first relevant instance. *Definition:* Equation (2.31). *Pages:* 27, 30, 136, 162–164

nDCG
: Normalized Discounted Cumulative Gain. IR metric that measures the quality of a ranking by considering non-binary relevance ratings. *Definition:* Equation (2.33). *Pages:* 27, 31, 134, 136, 140, 141

P@1
: Precision at 1. IR metric that measures the fraction of relevant documents in the top 1 retrieved document. *see* Prec@k. *Pages:* 136–138, 140, 141

P@10
: Precision at 10. IR metric that measures the fraction of relevant documents in the top 10 retrieved documents. *see* Prec@k. *Pages:* 136, 141

P@3
: Precision at 3. IR metric that measures the fraction of relevant documents in the top 3 retrieved documents. *see* Prec@k. *Pages:* 136, 138, 140, 141

Prec@k	Precision at k. IR metric that measures the fraction of relevant documents in the top k retrieved documents. *Definition:* Equation (2.27). *Pages:* 29, 30, 134
precision	Fraction of retrieved instances that are relevant. *Definition:* Equation (2.20). *Pages:* 27–29, 64, 66, 71
recall	Fraction of retrieved relevant instances. *Definition:* Equation (2.21). *Pages:* 27–29, 64, 66, 71
Recall@k	Recall at k. IR metric that measures the fraction of all relevant documents in the top k retrieved documents. *Definition:* Equation (2.28). *Pages:* 30
TF-IDF	Term Frequency-Inverse Document Frequency. A numerical statistic that reflects the importance of a word in a document relative to a collection of documents. *Definition:* Equation (2.47). *Pages:* 42, 43

9.4 Models

Absolute Tree Transformer — A transformer that uses absolute positional tree embeddings introduced by Shiv and Quirk (2019). *Pages:* 55, 62, 66, 71

Aroma — A non-neural CCS approach introduced by Luan et al. (2019). *Pages:* 115, 128, 139

Ast-attendgru — Model introduced by LeClair et al. (2019). *Pages:* 68

Attendgru — Model introduced by LeClair and McMillan (2019). *Pages:* 68

Bert — A transformer encoder LM introduced by Devlin et al. (2019). *Pages:* xiii, 39, 44, 50, 53, 54, 76, 81, 82, 86, 129

BiLSTM+GNN-LSTM — Model introduced by Fernandes et al. (2019). *Pages:* 68

bm25 — A industry standard probabilistic IR model introduced by Robertson and Zaragoza (2009). *Pages:* 7, 42–44, 124, 125, 135–138, 141, 145, 209, 231

Code+gnn+BiLSTM 2hops — Model introduced by LeClair et al. (2020). *Pages:* 68

Code2seq — A sequence-to-sequence code model that encodes pathes in ASTs introduced by Alon et al. (2019a). *Pages:* 50, 54, 66–68, 83

Code2vec — A path-based code embedding model introduced by Alon et al. (2019b). *Pages:* 50, 66, 115, 139

CodeBert — A transformer encoder code-LM pretrained with a discriminative objective introduced by Feng et al. (2020). *Pages:* 3, 39, 53, 54, 65, 68, 77, 82, 83, 87, 96, 111–113, 115, 116, 126, 128, 139, 178, 185, 196

CodeBuddy — The self-supervised CCS model described in Chapter 5 and introduced in **Villmow** et al. (2022). *Pages:* iv, vi, 4, 8, 10, 148–153, 158–167, 169, 171, 207–210, 217

CodeRetriever — A code retrieval model introduced by Li et al. (2022). *Pages:* 126, 127, 139

CodeT5 — A transformer encoder-decoder code-LM introduced by Wang et al. (2021b). *Pages:* 3, 14, 40, 78, 79, 83, 87, 90, 94, 104, 105, 111–113, 115, 116, 126, 139, 178, 197

ConvAttention — Introduced by Allamanis et al. (2016) as a model that combines convolutional and attentional mechanisms to encode code. *Pages:* 54, 66, 67

Conv-Seq2Seq — Model introduced by Gehring et al. (2017). *Pages:* 65

CORDER — A transformer encoder code-LM pretrained by augmenting code introduced by Bui et al. (2021). *Pages:* 127

CODEXT — A variant of the T5 model fine-tuned for code introduced by Phan et al. (2021). *Pages:* 83, 112, 113, 115, 116, 126, 139

DISCO — A transformer encoder code-LM pretrained on automatically created buggy and augmented versions of code introduced by Ding et al. (2022). *Pages:* 127, 139

DYNAMIC CONVOLUTION — Model introduced by Wu et al. (2019). *Pages:* 65

FASTTEXT — A word embedding model that uses sub-word information introduced by Bojanowski et al. (2017). *Pages:* 39, 179, 185, 187, 188, 202, 203

GPT — A transformer decoder LM introduced by Radford et al. (2018). *Pages:* 39, 40, 53, 81

GPT-2 — A transformer decoder LM introduced by Radford et al. (2019). *Pages:* 14, 82, 83

GRAPH2SEQ — Model introduced by (**DBLP:journals/corr/abs-1804-00823**). *Pages:* 68

GRAPHCODEBERT — A transformer encoder code-LM introduced by Guo et al. (2021). *Pages:* 83, 111, 112, 115, 116, 126, 139, 173, 194, 196, 199–203

HIERARCHICAL TRANSFORMER — Model introduced by Nguyen et al. (2020). *Pages:* 55, 65, 69, 73

INCODER — A transformer decoder code-LM that has been trained by infilling and autoregressive generation introduced by Fried et al. (2023). *Pages:* 9, 40, 173, 175, 178, 194, 196, 197, 199–203, 232

PLBART — A transformer encoder-decoder code-LM pretrained by denoising code introduced by Ahmad et al. (2021). *Pages:* 82, 111–113, 115, 116, 139

REGULARPT — Baseline for the SYNTAXPT model from Chapter 4 trained without syntax-aware objectives. *Pages:* 92, 93, 107, 110–113, 115

ROBERTA — A transformer encoder LM for NLP that is a variant of BERT introduced by Liu et al. (2019). *Pages:* 39, 53, 68, 76, 81, 82, 96, 97, 100

ROBERTA-CODE — A variant of the ROBERTA model fine-tuned for code on CODESEARCHNET by Feng et al. (2020). *Pages:* 82, 111–113, 115, 139

RST — Relative Structural Transformer. The relative structural transformer model presented in Chapter 3 and in **Villmow** et al. (2021b). *Pages:* 5, 6, 51, 62, 67–69, 72, 73, 75, 93, 110, 113, 207, 209

SynCobert — A transformer encoder code-LM pretrained with a contrastive objective in addition to a denoising objective introduced by Wang et al. (2021a). *Pages:* 83, 111, 115, 116, 126, 127, 139

SyntaxPT — The SyntaxPT model proposed in Chapter 7 and in **Villmow*** et al. (2023b). *Pages:* 193, 200–202

SyntaxPT — A transformer encoder-decoder code-LM pretrained with a self-supervised syntax-tree-aware multi-task objective introduced in Chapter 4. *Pages:* iii, iv, vi, 4, 6–9, 79, 80, 91–93, 107, 108, 110–113, 115–119, 129, 135, 136, 139, 141, 145, 146, 165, 171–175, 178, 179, 193, 194, 196–201, 203, 205, 207, 208

SyntaxPT-ccs — A transformer encoder-decoder code-LM pretrained with a self-supervised syntax-tree-aware multi-task objective introduced in Chapter 4. *Pages:* iii, iv, vi, 4, 7, 8, 137, 139–142, 145–148, 151–154, 157, 166, 207–210

Syntax-PT$_{\text{discriminative}}$ — A fine-tuned transformer encoder model proposed in Chapter 7 and in **Villmow*** et al. (2023b) based on the SyntaxPT model. *Pages:* 179, 200, 201

t5 — A transformer encoder-decoder LM for NLP introduced by Raffel et al. (2020). *Pages:* 15, 31, 33, 36, 39, 40, 77–79, 81, 83, 90, 93, 94, 112, 113, 115, 165

Transformer — This refers to the original transformer encoder-decoder architecture introduced by Vaswani et al. (2017), which is trained end-to-end without pretraining. *Pages:* 62, 65, 66, 68, 69, 71

Transformer (no PT) — A transformer baseline used in Chapter 4 that is trained end-to-end without pretraining. *Pages:* 92, 93, 110–113, 115

Tree2Seq — Model introduced by Shi et al. (2018). *Pages:* 65

TreeLSTM — A recursive neural network architecture that recursively encodes a tree by computing a node's representation based on its children using an LSTM unit. The architecture has been introduced by Tai et al. (2015). *Pages:* 54, 66

UniXcoder — A transformer code-LM introduced by Guo et al. (2022). *Pages:* 126, 127, 139

9.5 Terms

beam search — A search algorithm that explores the most likely paths in a search space by keeping track of the k most likely candidates at each step. Introduced in Section 2.2.2. *Pages:* 23, 24, 66, 102, 181

clone detection — Retrieval task that aims to detect code clones or duplicate code snippets in a codebase. It is part of the CodeXGlue benchmark. Introduced in Section 4.5.7. *Pages:* 7, 24, 25, 53, 101, 103, 105–109, 116, 120–122, 124, 125, 135, 139, 141, 164

cloze task — A type of language modeling task where a token is masked in a sentence, and the model is trained to predict the masked token. *Pages:* 37, 45, 122, 124, 125, 128, 129, 135, 137, 141, 142

code refinement — Sequence-to-sequence task that aims to automatically fix bugs in code snippets. It is part of the CodeXGlue benchmark. Introduced in Section 4.5.7. *Pages:* 101, 109

code summarization — Sequence-to-sequence task that aims to generate a concise description of a code snippet (e.g., for documentation or code search). It is part of the CodeXGlue benchmark. Introduced in Sections 3.4.3 and 4.5.7. *Pages:* 5, 6, 15, 23, 51, 53, 62–64, 67, 73, 75, 78, 79, 83, 88, 89, 92, 93, 101, 107, 110, 115

code translation — Sequence-to-sequence task that aims to translate code snippets from one programming language to another. It is part of the CodeXGlue benchmark. Introduced in Section 4.5.7. *Pages:* 90, 101, 102, 107, 109, 111, 118

dedenting — Deleaking step that probabilistically sets the indentation level of the code snippet to zero. This technique is introduced in Section 5.3.1. *Pages:* 165

defect detection — Classification task that aims to detect defects or bugs in code. It is part of the CodeXGlue benchmark. Introduced in Section 4.5.7. *Pages:* 7, 83, 101, 104, 106, 107, 116, 118, 125, 135, 139, 140

dropout — Regularization technique for neural networks that randomly sets a fraction of the input units to zero during training (usually around 10%). At test time, the output is scaled by the dropout rate. *Pages:* xiii, xv, 15, 21, 45, 66, 89, 94, 101, 105, 127

ElasticSearch — A distributed, RESTful search engine for text that uses the Okapi BM25 ranking function. https://www.elastic.co/elasticsearch/. *Pages:* 43, 137, 138

end-to-end
: This thesis refers to *end-to-end training* as training a model on a downstream task without pretraining on a large corpus of unlabeled data. *Pages:* 5, 43, 49, 51, 53, 60, 62, 63, 65, 68, 75, 76, 92, 101, 109, 110, 118, 127

GitHub
: A web-based platform for version control and collaboration using Git. It contains source code repositories, issue tracking, and project management tools. `https://github.com`. *Pages:* 3, 6, 16, 52, 64, 80, 82, 98, 99, 119, 120, 125, 146, 149, 161, 166, 172, 174, 178, 184, 188, 199

identifier deobfuscation
: Structural pretraining task for code understanding introduced by Lachaux et al. (2021) that replaces identifiers in a code snippet with unique tokens and trains a encoder-decoder model to predict the original identifiers. *Pages:* iii, 5, 6, 78–80, 85, 87, 88, 90, 94, 97, 117, 207

machine translation
: Automated process of translating text or speech from one language to another using computer algorithms. *Pages:* 18, 23, 28, 49, 51, 53, 62, 63, 65–68, 73

method naming
: Sequence-to-sequence task that aims to predict the method name for a code function. Introduced in Section 3.4.3. *Pages:* 5, 51, 62, 63, 67, 70, 73, 75, 79, 178, 189, 191, 194, 204

mutual identifier masking
: Deleaking step for a pair of code that analyzes the pairs ASTs to probabilistically mask identifiers that are shared between the two code snippets. This technique is introduced in Section 5.3.1. *Pages:* 165

n-gram
: A contiguous (sub-)sequence of n words or tokens. *Pages:* 23, 28, 29, 39, 64, 66, 82, 157, 174, 177, 178, 202, 217

repository
: A collection of source code files of the size of a software project or system. Typically, a repository is used to version control code projects for example using Git on GitHub. *Pages:* 2, 16, 52, 98, 99, 146, 149, 150, 164, 172, 174, 178, 197, 198

RESTful API
: A type of API that follows the principles of REST. *Pages:*

TENSORTREE
: Python library that enables developers to work with tree tensors in PyTorch, developed for this thesis and open-sourced at GitHub and PyPI (**Villmow** 2021). *Pages:* 6, 80, 97, 98, 131

tree-based file truncation
: Method to truncate a code file by removing large elements, based on its AST. This technique is introduced in Section 4.4.2. *Pages:* 6, 79, 80, 131, 132

tree-based span selection
: Method to sample a span of code-tokens by traversing its AST. This technique is introduced in Section 4.4.1, and adapted to CCS in Section 5.3.1. *Pages:* iii, 5, 6, 79, 80, 87, 88, 90, 91, 94, 117, 130, 154, 157, 165, 207

tree-sitter
: A parser generator tool and an incremental parsing library that builds ASTs and CSTs for code snippets (Brunsfeld 2023). *Pages:* 15, 16, 18, 63, 95, 99

Listing of Figures

Listing of Tables

References

Abadi, Martín, Ashish Agarwal, et al. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. In *CoRR* abs/1603.04467. arXiv: 1603.04467 (cited on page 100).

Abelson, Harold and Gerald J. Sussman (1985). Structure and Interpretation of Computer Programs. MIT Press. ISBN: 0-262-51036-7 (cited on page 13).

Ahmad, Wasi Uddin, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang (2021). Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pp. 2655–2668 (cited on pages 5, 85 sq., 241).

Allamanis, Miltiadis, Earl T. Barr, Christian Bird, and Charles Sutton (2014). 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*, pp. 281–293 (cited on pages 56, 186, 189, 216).

Allamanis, Miltiadis, Earl T. Barr, Christian Bird, and Charles Sutton (2015). 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*, pp. 38–49 (cited on pages 55, 190).

Allamanis, Miltiadis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton (2018). A Survey of Machine Learning for Big Code and Naturalness. In *ACM Comput. Surv.* 51.4, 81:1–81:37 (cited on pages 55, 189).

Allamanis, Miltiadis, Hao Peng, and Charles Sutton (2016). 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*. Vol. 48. JMLR Workshop and Conference Proceedings, pp. 2091–2100 (cited on pages 56, 66, 240).

Allamanis, Miltiadis and Charles Sutton (2013). 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*, pp. 207–216 (cited on page 84).

Alon, Uri, Shaked Brody, Omer Levy, and Eran Yahav (2019a). code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019* (cited on pages 5, 17, 52 sq., 55 sq., 64 sqq., 69, 237, 240).

Alon, Uri, Meital Zilberstein, Omer Levy, and Eran Yahav (2019b). code2vec: learning distributed representations of code. In *Proc. ACM Program. Lang.* 3.POPL, 40:1–40:29 (cited on pages 17, 52, 240).

Amershi, Saleema, Andrew Begel, et al. (2019). Software engineering for machine learning: a case study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 291–300 (cited on page 2).

Andaloussi, Amine Abbad, Thierry Sorg, and Barbara Weber (2022). Estimating developers' cognitive load at a fine-grained level using eye-tracking measures. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*, pp. 111–121 (cited on page 188).

Andriantiana, Eric O. D., Kenneth Dadedzi, and Stephan Wagner (2018). The ancestral matrix of a rooted tree. In *math* abs/1809.03364. arXiv: 1809.03364 [math.CO] (cited on page 61).

Arnaoudova, Venera, Massimiliano Di Penta, and Giuliano Antoniol (2016). Linguistic antipatterns: what they are and how developers perceive them. In *Empir. Softw. Eng.* 21.1, pp. 104–158 (cited on pages 184 sq., 188, 202 sq.).

Ba, Lei Jimmy, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). Layer Normalization. In *CoRR* abs/1607.06450. arXiv: 1607.06450 (cited on page 35).

Babii, Hlib, Andrea Janes, and Romain Robbes (2019). Modeling Vocabulary for Big Code Machine Learning. In *CoRR* abs/1904.01873. arXiv: 1904.01873 (cited on page 65).

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2015). 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* (cited on page 55).

Bajracharya, Sushil Krishna and Cristina Videira Lopes (2012). Analyzing and mining a code search engine usage log. In *Empir. Softw. Eng.* 17.4-5, pp. 424–466 (cited on pages 161 sq.).

Bajracharya, Sushil Krishna, Joel Ossher, and Cristina Videira Lopes (2010). Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pp. 157–166 (cited on pages 44, 135).

Bardes, Adrien, Jean Ponce, and Yann LeCun (2022). VICReg: Variance-Invariance-Covariance Regularization for Self-Supervised Learning. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022* (cited on pages 28 sq.).

Bavishi, Rohan, Michael Pradel, and Koushik Sen (2018). Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. In *CoRR* abs/1809.05193. arXiv: 1809.05193 (cited on page 190).

Baxter, Ira D., Andrew Yahin, et al. (1998). Clone Detection Using Abstract Syntax Trees. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pp. 368–377 (cited on page 55).

Bengio, Yoshua, Réjean Ducharme, and Pascal Vincent (2000). A Neural Probabilistic Language Model. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*, pp. 932–938 (cited on page 47).

Bichsel, Benjamin, Veselin Raychev, Petar Tsankov, and Martin T. Vechev (2016). Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference*

on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pp. 343–355 (cited on page 55).

Bielik, Pavol, Veselin Raychev, and Martin T. Vechev (2016). 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*. Vol. 48. JMLR Workshop and Conference Proceedings, pp. 2933–2942 (cited on page 55).

Binder, Felix, **Johannes Villmow**, and Adrian Ulges (2020). Bidirectional Transformer Language Models for Smart Autocompletion of Source Code. In *50. Jahrestagung der Gesellschaft für Informatik, INFORMATIK 2020 - Back to the Future, Karlsruhe, Germany, 28. September - 2. Oktober 2020*. Vol. P-307. LNI, pp. 915–922 (cited on page 10).

Bojanowski, Piotr, Edouard Grave, Armand Joulin, and Tomás Mikolov (2017). Enriching Word Vectors with Subword Information. In *Trans. Assoc. Comput. Linguistics* 5, pp. 135–146 (cited on pages 41, 191, 197, 199, 241).

Borges, Hudson and Marco Túlio Valente (2018). What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. In *J. Syst. Softw.* 146, pp. 112–129 (cited on page 102).

Bromley, Jane, Isabelle Guyon, et al. (1993). Signature Verification Using a Siamese Time Delay Neural Network. In *Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]*, pp. 737–744 (cited on pages 27 sq.).

Brooks, Ruven E. (1983). Towards a Theory of the Comprehension of Computer Programs. In *Int. J. Man Mach. Stud.* 18.6, pp. 543–554 (cited on pages 80, 184, 188).

Brown, Tom B., Benjamin Mann, et al. (2020). Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (cited on pages 43, 79, 127).

Bui, Nghi D. Q., Yijun Yu, and Lingxiao Jiang (2021). Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, pp. 511–521 (cited on pages 135, 241).

Buse, Raymond P. L. and Westley Weimer (2010). Learning a Metric for Code Readability. In *IEEE Trans. Software Eng.* 36.4, pp. 546–558 (cited on page 189).

Butler, Simon, Michel Wermelinger, Yijun Yu, and Helen Sharp (2010). Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *14th European Conference on Software Maintenance and Reengineering, CSMR 2010, 15-18 March 2010, Madrid, Spain*, pp. 156–165 (cited on pages 184 sq., 188 sq., 202 sq.).

Canny, John F. (1986). A Computational Approach to Edge Detection. In *IEEE Trans. Pattern Anal. Mach. Intell.* 8.6, pp. 679–698 (cited on page 171).

Caprile, Bruno and Paolo Tonella (2000). Restructuring Program Identifier Names. In *2000 International Conference on Software Maintenance, ICSM 2000, San Jose, California, USA, October 11-14, 2000*, pp. 97–107 (cited on pages 184, 189).

Cettolo, Mauro, Jan Niehues, et al. (2014). Report on the 11th IWSLT evaluation campaign. In *Proceedings of the 11th International Workshop on Spoken Language Translation: Evaluation Campaign@IWSLT 2014, Lake Tahoe, CA, USA, December 4-5, 2014* (cited on pages 67, 237).

Chatterjee, Shaunak, Sudeep Juvekar, and Koushik Sen (2009). SNIFF: A Search Engine for Java Using Free-Form Queries. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Vol. 5503. Lecture Notes in Computer Science, pp. 385–400 (cited on pages 44, 135).

Chen, Fang, Jianlong Zhou, et al. (2016). Robust Multimodal Cognitive Load Measurement. Human-Computer Interaction Series. Springer. ISBN: 978-3-319-31698-7 (cited on page 188).

Chen, Mark, Jerry Tworek, et al. (2021). Evaluating Large Language Models Trained on Code. In *CoRR* abs/2107.03374. arXiv: 2107.03374 (cited on pages 3, 43, 127, 184).

Chen, Qibin, Jeremy Lacomis, et al. (2022). VarCLR: Variable Semantic Representation Pre-training via Contrastive Learning. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 2327–2339 (cited on pages 187, 190, 212).

Chen, Ting, Simon Kornblith, Mohammad Norouzi, and Geoffrey E. Hinton (2020). A Simple Framework for Contrastive Learning of Visual Representations. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research, pp. 1597–1607 (cited on pages 28 sq.).

Ciniselli, Matteo, Nathan Cooper, et al. (2022). An Empirical Study on the Usage of Transformer Models for Code Completion. In *IEEE Trans. Software Eng.* 48.12, pp. 4818–4837 (cited on pages 83, 190).

Clark, Kevin, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning (2019). What Does BERT Look at? An Analysis of BERT's Attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@ACL 2019, Florence, Italy, August 1, 2019*, pp. 276–286 (cited on page 138).

Clark, Kevin, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning (2020). ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020* (cited on pages 79, 83, 197).

Conneau, Alexis, Kartikay Khandelwal, et al. (2020). Unsupervised Cross-lingual Representation Learning at Scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pp. 8440–8451 (cited on pages 16, 130).

Corbo, Filippo, Concettina Del Grosso, and Massimiliano Di Penta (2007). Smart Formatter: Learning Coding Style from Existing Source Code. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*, pp. 525–526 (cited on page 189).

Craswell, Nick, Bhaskar Mitra, et al. (2020). Overview of the TREC 2019 deep learning track. In *CoRR* abs/2003.07820. arXiv: 2003.07820 (cited on pages 166, 177).

Dahal, Samip, Adyasha Maharana, and Mohit Bansal (2022). Scotch: A Semantic Code Search Engine for IDEs. In *Deep Learning for Code Workshop* (cited on pages 129, 137, 159).

Dam, Hoa Khanh, Truyen Tran, John C. Grundy, and Aditya K. Ghose (2016). DeepSoft: a vision for a deep model of software. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pp. 944–947 (cited on page 84).

Dauphin, Yann N., Angela Fan, Michael Auli, and David Grangier (2017). Language Modeling with Gated Convolutional Networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Vol. 70. Proceedings of Machine Learning Research, pp. 933–941 (cited on page 54).

Deerwester, Scott C., Susan T. Dumais, et al. (1990). Indexing by Latent Semantic Analysis. In *J. Am. Soc. Inf. Sci.* 41.6, pp. 391–407 (cited on page 46).

Deissenboeck, Florian and Markus Pizka (2006). Concise and consistent naming. In *Softw. Qual. J.* 14.3, pp. 261–282 (cited on pages 2, 187, 189).

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pp. 4171–4186 (cited on pages 3, 16, 41, 55, 78 sq., 86, 113, 240).

Dhar, Payal (2020). The carbon impact of artificial intelligence. In *Nat. Mach. Intell.* 2.8, pp. 423–425 (cited on page 128).

Ding, Yangruibo, Luca Buratti, et al. (2022). Towards Learning (Dis)-Similarity of Source Code from Program Contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pp. 6300–6312 (cited on pages 136, 241).

Dohmke, Thomas, Marco Iansiti, and Greg Richards (2023). Sea Change in Software Development: Economic and Productivity Analysis of the AI-Powered Developer Lifecycle. In *CoRR* abs/2306.15033. arXiv:2306.15033 [econ, q-fin] (cited on page 161).

Dong, Li, Nan Yang, et al. (2019). Unified Language Model Pre-training for Natural Language Understanding and Generation. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 13042–13054 (cited on page 83).

Douze, Matthijs, Alexandr Guzhva, et al. (2024). The Faiss library. In *CoRR* abs/2401.08281. arXiv: 2401.08281 (cited on pages 27, 165).

Dubey, Abhimanyu, Abhinav Jauhri, et al. (2024). The Llama 3 Herd of Models. In *CoRR* abs/2407.21783. arXiv: 2407.21783 (cited on page 43).

Fakhoury, Sarah, Devjeet Roy, et al. (2020). Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. In *Empir. Softw. Eng.* 25.3, pp. 2140–2178 (cited on pages 2, 80, 184, 187 sq.).

Falleri, Jean-Rémy, Floréal Morandat, et al. (2014). Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pp. 313–324 (cited on page 17).

Fan, Angela, David Grangier, and Michael Auli (2018). Controllable Abstractive Summarization. In *Proceedings of the 2nd Workshop on Neural Machine Translation and Generation, NMT@ACL 2018, Melbourne, Australia, July 20, 2018*, pp. 45–54 (cited on pages 51, 55).

Feng, Zhangyin, Daya Guo, et al. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*. Vol. EMNLP 2020. Findings of ACL, pp. 1536–1547 (cited on pages 3, 5, 41, 55 sq., 67, 70, 79, 81, 84, 113, 119, 125, 240 sq.).

Fernandes, Patrick, Miltiadis Allamanis, and Marc Brockschmidt (2019). Structured Neural Summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019* (cited on pages 56, 240).

Ferrante, Jeanne, Karl J. Ottenstein, and Joe D. Warren (1987). The Program Dependence Graph and Its Use in Optimization. In *ACM Trans. Program. Lang. Syst.* 9.3, pp. 319–349 (cited on page 17).

Firth, J. (1957). A Synopsis of Linguistic Theory 1930-1955. In *Studies in Linguistic Analysis* (cited on page 39).

Fischer, Gerhard, Scott Henninger, and David F. Redmiles (1991). Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13th International Conference on Software Engineering, Austin, TX, USA, May 13-17, 1991*, pp. 318–328 (cited on page 134).

Fowler, Martin (1999). Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series. Addison-Wesley. ISBN: 978-0-201-48567-7 (cited on page 51).

Frakes, William B. and Christopher J. Fox (1996). Quality Improvement Using A Software Reuse Failure Modes Model. In *IEEE Trans. Software Eng.* 22.4, pp. 274–279 (cited on page 128).

Frakes, William B. and Brian A. Nejmeh (1987). Software Reuse Through Information Retrieval. In *COMPCON'87, Digest of Papers, Thirty-Second IEEE Computer Society International Conference, San Francisco, California, USA, February 23-27, 1987*, pp. 380–384 (cited on pages 2, 134).

Franks, Christine, Zhaopeng Tu, Premkumar T. Devanbu, and Vincent J. Hellendoorn (2015). 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*, pp. 705–708 (cited on page 84).

Fried, Daniel, Armen Aghajanyan, et al. (2023). InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023* (cited on pages 42, 190, 208, 210, 213, 241).

Fritz, Thomas, Andrew Begel, et al. (2014). Using psycho-physiological measures to assess task difficulty in software development. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pp. 402–413 (cited on page 188).

Gao, Leo, Stella Biderman, et al. (2021a). The Pile: An 800GB Dataset of Diverse Text for Language Modeling. In *CoRR* abs/2101.00027. arXiv: 2101.00027 (cited on page 3).

Gao, Tianyu, Xingcheng Yao, and Danqi Chen (2021b). SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pp. 6894–6910 (cited on pages 28 sq., 47, 136, 138).

Gehring, Jonas, Michael Auli, et al. (2017). Convolutional Sequence to Sequence Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Vol. 70. Proceedings of Machine Learning Research, pp. 1243–1252 (cited on pages 54 sq., 67, 240).

Gellenbeck, Edward M and Curtis R Cook (1991). An investigation of procedure and variable names as beacons during program comprehension. In *Empirical studies of programmers: Fourth workshop*. Ablex Publishing, Norwood, NJ, pp. 65–81 (cited on page 188).

Gillick, Daniel, Alessandro Presta, and Gaurav Singh Tomar (2018). End-to-End Retrieval in Continuous Space. In *CoRR* abs/1811.08008. arXiv: 1811.08008 (cited on page 46).

Goldberg, Yoav (2016). A Primer on Neural Network Models for Natural Language Processing. In *J. Artif. Intell. Res.* 57, pp. 345–420 (cited on page 54).

Goodfellow, Ian J., Yoshua Bengio, and Aaron C. Courville (2016). Deep Learning. Adaptive computation and machine learning. MIT Press. ISBN: 978-0-262-03561-3 (cited on pages xvii, 11, 13, 21–24, 26, 39, 157, 197, 238).

Goodfellow, Ian J., Mehdi Mirza, et al. (2014). An Empirical Investigation of Catastrophic Forgeting in Gradient-Based Neural Networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings* (cited on page 104).

Grazia, Luca Di and Michael Pradel (2023). Code Search: A Survey of Techniques for Finding Code. In *ACM Comput. Surv.* 55.11, 220:1–220:31 (cited on pages 161 sq.).

Grechanik, Mark, Kevin M. Conroy, and Katharina Probst (2007). Finding Relevant Applications for Prototyping. In *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*, p. 12 (cited on pages 2, 44, 134 sq.).

Grechanik, Mark and Denys Poshyvanyk (2008). Evaluating recommended applications. In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering, RSSE 2008, Atlanta, GA, USA, November 9, 2008*, pp. 33–35 (cited on pages 44, 135).

Gu, Xiaodong, Hongyu Zhang, and Sunghun Kim (2018). Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 933–944 (cited on pages 51, 55).

Guo, Daya, Shuai Lu, et al. (2022). UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pp. 7212–7225 (cited on pages 86, 121, 135 sq., 149, 242).

Guo, Daya, Shuo Ren, et al. (2021). GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021* (cited on pages 17, 52, 85, 113, 135, 208 sq., 241).

Guu, Kelvin, Kenton Lee, et al. (2020). REALM: Retrieval-Augmented Language Model Pre-Training. In *CoRR* abs/2002.08909. arXiv: 2002.08909 (cited on page 47).

He, Kaiming, Haoqi Fan, et al. (2020). Momentum Contrast for Unsupervised Visual Representation Learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pp. 9726–9735 (cited on pages 28 sq., 152).

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016). Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 770–778 (cited on page 35).

Hellendoorn, Vincent J. and Premkumar T. Devanbu (2017). 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*, pp. 763–773 (cited on pages 56, 84).

Hevner, Alan, Alan R, et al. (2004). Design Science in Information Systems Research. In *Management Information Systems Quarterly* 28, pp. 75– (cited on pages 11, 160).

Hilton, Peter and Felienne Hermans (2017). Naming Guidelines for Professional Programmers. In *Proceedings of the 28th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2017, Delft, The Netherlands, July 1-3, 2017*, p. 19 (cited on pages 184 sq., 188, 202 sq.).

Hindle, Abram, Earl T. Barr, et al. (2012). On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pp. 837–847 (cited on pages 3, 84, 189 sq.).

Hochreiter, Sepp and Jürgen Schmidhuber (1997). Long Short-Term Memory. In *Neural Comput.* 9.8, pp. 1735–1780 (cited on page 54).

Hofmann, Thomas (1999). Probabilistic Latent Semantic Indexing. In *SIGIR '99: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 15-19, 1999, Berkeley, CA, USA*, pp. 50–57 (cited on page 46).

Holmes, Reid and Gail C. Murphy (2005). Using structural context to recommend source code examples. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pp. 117–125 (cited on page 136).

Holmes, Reid, Robert J. Walker, and Gail C. Murphy (2005). Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pp. 237–240 (cited on page 136).

Holtzman, Ari, Jan Buys, et al. (2020). The Curious Case of Neural Text Degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020* (cited on page 25).

Hosang, Jan Hendrik, Rodrigo Benenson, and Bernt Schiele (2017). Learning Non-maximum Suppression. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pp. 6469–6477 (cited on page 171).

Hu, Xing, Ge Li, et al. (2018). Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pp. 200–210 (cited on page 56).

Hu, Yaojie, Xingjian Shi, Qiang Zhou, and Lee Pike (2022). Fix Bugs with Transformer through a Neural-Symbolic Edit Grammar. In *CoRR* abs/2204.06643. arXiv: 2204.06643 (cited on page 118).

Huang, Gao, Chuan Guo, et al. (2016). Supervised Word Mover's Distance. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 4862–4870 (cited on page 54).

Huang, Lei, Weijiang Yu, et al. (2023). A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. In *CoRR* abs/2311.05232. arXiv: 2311.05232 (cited on pages 3, 177).

Huang, Po-Sen, Xiaodong He, et al. (2013). Learning deep structured semantic models for web search using clickthrough data. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pp. 2333–2338 (cited on page 46).

Humeau, Samuel, Kurt Shuster, Marie-Anne Lachaux, and Jason Weston (2020). Poly-encoders: Architectures and Pre-training Strategies for Fast and Accurate Multi-sentence Scoring. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020* (cited on page 46).

Husain, Hamel, Ho-Hsiang Wu, et al. (2019). CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. In *CoRR* abs/1909.09436. arXiv: 1909.09436 (cited on pages 67, 104, 107, 128, 133, 135, 144, 237).

Izacard, Gautier, Mathilde Caron, et al. (2022). Unsupervised Dense Information Retrieval with Contrastive Learning. In *Trans. Mach. Learn. Res.* 2022 (cited on page 152).

Izacard, Gautier and Edouard Grave (2021). Distilling Knowledge from Reader to Retriever for Question Answering. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021* (cited on page 47).

Izacard, Gautier, Fabio Petroni, et al. (2020). A Memory Efficient Baseline for Open Domain Question Answering. In *CoRR* abs/2012.15156. arXiv: 2012.15156 (cited on page 152).

Jablonski, Patricia and Daqing Hou (2007). CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange, ETX 2007, Montreal, Quebec, Canada, October 21, 2007*, pp. 16–20 (cited on page 189).

Jain, Paras, Ajay Jain, et al. (2021). Contrastive Code Representation Learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pp. 5954–5971 (cited on page 135).

Jiang, Xue, Zhuoran Zheng, et al. (2021). TreeBERT: A tree-based pre-trained model for programming language. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial*

Intelligence, UAI 2021, Virtual Event, 27-30 July 2021. Vol. 161. Proceedings of Machine Learning Research, pp. 54–63 (cited on page 85).

Jones, Karen Sparck, Steve Walker, and Stephen E. Robertson (2000). A probabilistic model of information retrieval: development and comparative experiments - Part 1. In *Inf. Process. Manag.* 36.6, pp. 779–808 (cited on pages 147 sq.).

Joulin, Armand, Edouard Grave, et al. (2016). FastText.zip: Compressing text classification models. In *CoRR* abs/1612.03651. arXiv: 1612.03651 (cited on page 199).

Jurafsky, Dan and James H. Martin (2009). Speech and Language Processing: an Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 2nd Edition. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International. ISBN: 9780135041963 (cited on pages 13, 15, 25 sq., 35, 42, 45, 192).

Kanade, Aditya, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi (2020). Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research, pp. 5110–5121 (cited on page 84).

Kaplan, Jared, Sam McCandlish, et al. (2020). Scaling Laws for Neural Language Models. In *CoRR* abs/2001.08361. arXiv: 2001.08361 (cited on page 127).

Karpukhin, Vladimir, Barlas Oguz, et al. (2020). Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pp. 6769–6781 (cited on pages 27, 46).

Kim, Kisub, Dongsun Kim, et al. (2018). FaCoY: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 946–957 (cited on page 137).

Kim, Seohyun, Jinman Zhao, Yuchi Tian, and Satish Chandra (2021). Code Prediction by Feeding Trees to Transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pp. 150–162 (cited on page 56).

Kim, Yoon (2014). Convolutional Neural Networks for Sentence Classification. 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*, pp. 1746–1751 (cited on page 54).

Kingma, Diederik P. and Jimmy Ba (2015). 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* (cited on pages 22, 68).

Kipf, Thomas N. and Max Welling (2017). Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings* (cited on page 54).

Kitaev, Nikita, Lukasz Kaiser, and Anselm Levskaya (2020). Reformer: The Efficient Transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020* (cited on page 222).

Ko, Amy J., Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung (2006). An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. In *IEEE Trans. Software Eng.* 32.12, pp. 971–987 (cited on pages 161 sq.).

Kudo, Taku (2018). Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. 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*, pp. 66–75 (cited on pages 16 sq.).

Kudo, Taku and John Richardson (2018). SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, pp. 66–71 (cited on page 17).

Kusupati, Aditya, Gantavya Bhatt, et al. (2022). Matryoshka Representation Learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022* (cited on page 150).

Lachaux, Marie-Anne, Baptiste Rozière, Marc Szafraniec, and Guillaume Lample (2021). DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 14967–14979 (cited on pages 6, 79 sq., 82, 86, 88, 91 sq., 123, 244).

Lacomis, Jeremy, Pengcheng Yin, et al. (2019). DIRE: A Neural Approach to Decompiled Identifier Naming. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pp. 628–639 (cited on page 190).

Lan, Zhenzhong, Mingda Chen, et al. (2020). ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020* (cited on page 83).

Lawrie, Dawn J., Christopher Morrell, Henry Feild, and David W. Binkley (2006). What's in a Name? A Study of Identifiers. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pp. 3–12 (cited on pages 187 sq.).

LeClair, Alexander, Sakib Haque, Lingfei Wu, and Collin McMillan (2020). Improved Code Summarization via a Graph Neural Network. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pp. 184–195 (cited on pages 56, 70, 240).

LeClair, Alexander, Siyuan Jiang, and Collin McMillan (2019). A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 795–806 (cited on pages 56, 240).

LeClair, Alexander and Collin McMillan (2019). Recommendations for Datasets for Source Code Summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pp. 3931–3937 (cited on pages 66 sq., 70, 237, 240).

LeCun, Yann, Yoshua Bengio, and Geoffrey E. Hinton (2015). Deep learning. In *Nat.* 521.7553, pp. 436–444 (cited on page 54).

Lee, Kenton, Ming-Wei Chang, and Kristina Toutanova (2019). Latent Retrieval for Weakly Supervised Open Domain Question Answering. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pp. 6086–6096 (cited on pages 7, 47, 130, 138, 140).

Lewis, Mike, Marjan Ghazvininejad, et al. (2020a). Pre-training via Paraphrasing. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (cited on page 47).

Lewis, Mike, Yinhan Liu, et al. (2020b). BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pp. 7871–7880 (cited on pages 43, 78, 83, 91).

Li, Guangjie, Hui Liu, and Ally S. Nyamawe (2021). A Survey on Renamings of Software Entities. In *ACM Comput. Surv.* 53.2, 41:1–41:38 (cited on pages 184, 188).

Li, Jian, Yue Wang, Michael R. Lyu, and Irwin King (2018). 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*, pp. 4159–4165 (cited on page 84).

Li, Raymond, Loubna Ben Allal, et al. (2023). StarCoder: may the source be with you! In *Trans. Mach. Learn. Res.* 2023 (cited on pages 17, 42, 119).

Li, Xiaonan, Yeyun Gong, et al. (2022). CodeRetriever: A Large Scale Contrastive Pre-Training Method for Code Search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pp. 2898–2910 (cited on pages 135 sq., 240).

Likert, Rensis (1932). A Technique for the Measurement of Attitudes. In *Archives of Psychology* 140, pp. 1–55 (cited on page 175).

Lin, Bin, Simone Scalabrino, et al. (2017). Investigating the Use of Code Analysis and NLP to Promote a Consistent Usage of Identifiers. In *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017*, pp. 81–90 (cited on pages 184, 189).

Linstead, Erik, Sushil Krishna Bajracharya, et al. (2009). Sourcerer: mining and searching internet-scale software repositories. In *Data Min. Knowl. Discov.* 18.2, pp. 300–336 (cited on page 135).

Liu, Fang, Ge Li, Yunfei Zhao, and Zhi Jin (2020). Multi-task Learning based Pre-trained Language Model for Code Completion. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pp. 473–485 (cited on page 84).

Liu, Fang, Ge Li, et al. (2022). Learning to Recommend Method Names with Global Context. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 1294–1306 (cited on page 190).

Liu, Jiacheng, Sewon Min, et al. (2024). Infini-gram: Scaling Unbounded n-gram Language Models to a Trillion Tokens. In *CoRR* abs/2401.17377. arXiv: 2401.17377 (cited on page 41).

Liu, Shangqing, Bozhi Wu, et al. (2023). ContraBERT: Enhancing Code Pre-trained Models via Contrastive Learning. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pp. 2476–2487 (cited on pages 121, 135 sq., 149).

Liu, Yinhan, Myle Ott, et al. (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach. In *CoRR* abs/1907.11692. arXiv: 1907.11692 (cited on pages 41, 55, 70, 79, 83, 99, 103, 114, 241).

Loshchilov, Ilya and Frank Hutter (2019). Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019* (cited on pages 22, 105, 146).

Lu, Shuai, Nan Duan, et al. (2022). ReACC: A Retrieval-Augmented Code Completion Framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pp. 6227–6240 (cited on pages 133, 137).

Lu, Shuai, Daya Guo, et al. (2021). CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual* (cited on pages 6, 82, 94, 103–106, 108 sq., 115, 123, 127, 134, 149, 237).

Luan, Sifei, Di Yang, et al. (2019). Aroma: code recommendation via structural code search. In *Proc. ACM Program. Lang.* 3.OOPSLA, 152:1–152:28 (cited on pages 137, 240).

Luan, Yi, Jacob Eisenstein, Kristina Toutanova, and Michael Collins (2021). Sparse, Dense, and Attentional Representations for Text Retrieval. In *Trans. Assoc. Comput. Linguistics* 9, pp. 329–345 (cited on page 46).

Lv, Fei, Hongyu Zhang, et al. (2015). CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pp. 260–270 (cited on pages 44, 135).

Malkov, Yury A. and Dmitry A. Yashunin (2020). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. In *IEEE Trans. Pattern Anal. Mach. Intell.* 42.4, pp. 824–836 (cited on pages 27, 165, 170).

Mani, Senthil, Anush Sankaran, and Rahul Aralikatte (2019). DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, COMAD/CODS 2019, Kolkata, India, January 3-5, 2019*, pp. 171–179 (cited on pages 51, 55).

Mann, H. B. and D. R. Whitney (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. In *The Annals of Mathematical Statistics* 18.1, pp. 50–60 (cited on page 175).

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze (2008). Introduction to Information Retrieval. Cambridge University Press. ISBN: 978-0-521-86571-5 (cited on pages 13, 32 sq., 44).

Manning, Christopher D., Mihai Surdeanu, et al. (2014). The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations*, pp. 55–60 (cited on page 67).

Mastropaolo, Antonio, Emad Aghajani, Luca Pascarella, and Gabriele Bavota (2022). Automated Variable Renaming: Are We There Yet? In *CoRR* abs/2212.05738. arXiv: 2212.05738 (cited on page 185).

Mastropaolo, Antonio, Emad Aghajani, Luca Pascarella, and Gabriele Bavota (2023). Automated variable renaming: are we there yet? In *Empir. Softw. Eng.* 28.2, p. 45 (cited on page 190).

Maynez, Joshua, Shashi Narayan, Bernd Bohnet, and Ryan T. McDonald (2020). On Faithfulness and Factuality in Abstractive Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pp. 1906–1919 (cited on page 128).

McCann, Bryan, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher (2018). The Natural Language Decathlon: Multitask Learning as Question Answering. In *CoRR* abs/1806.08730. arXiv: 1806.08730 (cited on page 54).

Mielke, Sabrina J., Zaid Alyafeai, et al. (2021). Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP. In *CoRR* abs/2112.10508. arXiv: 2112.10508 (cited on page 17).

Mikolov, Tomás, Kai Chen, Greg Corrado, and Jeffrey Dean (2013a). Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings* (cited on pages 40, 54).

Mikolov, Tomás, Edouard Grave, et al. (2018). Advances in Pre-Training Distributed Word Representations. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018, Miyazaki, Japan, May 7-12, 2018* (cited on page 54).

Mikolov, Tomás, Ilya Sutskever, et al. (2013b). Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pp. 3111–3119 (cited on pages 27, 47, 62, 199).

Mikolov, Tomás, Wen-tau Yih, and Geoffrey Zweig (2013c). Linguistic Regularities in Continuous Space Word Representations. 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*, pp. 746–751 (cited on pages 40, 54).

Mishne, Alon, Sharon Shoham, and Eran Yahav (2012). Typestate-based semantic code search over partial programs. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pp. 997–1016 (cited on pages 129, 136).

Mitra, Bhaskar and Nick Craswell (2018). An Introduction to Neural Information Retrieval. In *Found. Trends Inf. Retr.* 13.1, pp. 1–126 (cited on pages 44, 46).

Mou, Lili, Ge Li, et al. (2016). 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*, pp. 1287–1293 (cited on pages 104, 109 sq.).

Mukherjee, Rohan, Chris Jermaine, and Swarat Chaudhuri (2020). Searching a Database of Source Codes Using Contextualized Code Search. In *Proc. VLDB Endow.* 13.10, pp. 1765–1778 (cited on pages 7, 129, 137, 159).

Murali, Vijayaraghavan, Letao Qi, Swarat Chaudhuri, and Chris Jermaine (2018). Neural Sketch Learning for Conditional Program Generation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings* (cited on page 137).

Neelakantan, Arvind, Tao Xu, et al. (2022). Text and Code Embeddings by Contrastive Pre-Training. In *CoRR* abs/2201.10005. arXiv: 2201.10005 (cited on pages 150 sq.).

Nguyen, Anh Tuan and Tien N. Nguyen (2015). Graph-Based Statistical Language Model for Code. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pp. 858–868 (cited on page 84).

Nguyen, Tri, Mir Rosenberg, et al. (2016). MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. In *Proceedings of the Workshop on Cognitive Computation: Integrating neural and symbolic approaches 2016 co-located with the 30th Annual Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain, December 9, 2016*. Vol. 1773. CEUR Workshop Proceedings (cited on pages 46, 130).

Nguyen, Xuan-Phi, Shafiq R. Joty, Steven C. H. Hoi, and Richard Socher (2020). Tree-Structured Attention with Hierarchical Accumulation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020* (cited on pages 52, 57, 67 sq., 71 sq., 241).

Nogueira, Rodrigo Frassetto, Wei Yang, Kyunghyun Cho, and Jimmy Lin (2019). Multi-Stage Document Ranking with BERT. In *CoRR* abs/1910.14424. arXiv: 1910.14424 (cited on page 46).

Oord, Aäron van den, Yazhe Li, and Oriol Vinyals (2018). Representation Learning with Contrastive Predictive Coding. In *CoRR* abs/1807.03748. arXiv: 1807.03748 (cited on page 28).

OpenAI (2023). GPT-4 Technical Report. In *CoRR* abs/2303.08774. arXiv: 2303.08774 (cited on page 43).

Ott, Myle, Sergey Edunov, et al. (2019). fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Demonstrations*, pp. 48–53 (cited on page 68).

Ouyang, Long, Jeffrey Wu, et al. (2022). Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on*

Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022 (cited on page 43).

Pan, Bing, Helene Hembrooke, et al. (2007). In Google We Trust: Users' Decisions on Rank, Position, and Relevance. In *J. Comput. Mediat. Commun.* 12.3, pp. 801–823 (cited on page 170).

Panchenko, Oleksandr, Hasso Plattner, and Alexander Zeier (2011). What do developers search for in source code and why. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. SUITE '11. Waikiki, Honolulu, HI, USA, pp. 33–36. ISBN: 9781450305976 (cited on page 162).

Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu (2002). Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pp. 311–318 (cited on page 31).

Parvez, Md. Rizwan, Wasi Uddin Ahmad, et al. (2021). Retrieval Augmented Code Generation and Summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pp. 2719–2734 (cited on page 133).

Paszke, Adam, Sam Gross, et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035 (cited on pages 22, 54, 68, 82, 100, 103 sq.).

Paul, Santanu and Atul Prakash (1994). A Framework for Source Code Search Using Program Patterns. In *IEEE Trans. Software Eng.* 20.6, pp. 463–475 (cited on page 17).

Paulus, Romain, Caiming Xiong, and Richard Socher (2018). A Deep Reinforced Model for Abstractive Summarization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings* (cited on page 25).

Peng, Sida, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer (2023). The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. In *CoRR* abs/2302.06590. arXiv: 2302.06590 (cited on page 161).

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning (2014). Glove: Global Vectors for Word Representation. 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*, pp. 1532–1543 (cited on pages 47, 54).

Peters, Matthew E., Mark Neumann, et al. (2018). Deep Contextualized Word Representations. 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)*, pp. 2227–2237 (cited on pages 41, 83).

Phan, Long N., Hieu Tran, et al. (2021). CoTexT: Multi-task Learning with Code-Text Transformer. In *CoRR* abs/2105.08645. arXiv: 2105.08645 (cited on pages 85 sq., 118, 135, 241).

Radford, Alec, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever (2018). Improving language understanding by generative pre-training. en. In *OpenAI blog* (cited on pages 3, 22, 41 sq., 55, 78 sq., 241).

Radford, Alec, Jeffrey Wu, et al. (2019). Language Models are Unsupervised Multitask Learners. In *OpenAI blog* 1.8, p. 9 (cited on pages 17, 79, 99, 241).

Raffel, Colin, Noam Shazeer, et al. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. In *J. Mach. Learn. Res.* 21, 140:1–140:67 (cited on pages 17, 33, 35, 37 sq., 41, 43, 79 sq., 83, 85 sq., 88, 91, 93, 96, 103, 242).

Rahman, Md. Masudur, Jed Barson, et al. (2018). Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pp. 465–475 (cited on page 162).

Ram, Ori, Yoav Levine, et al. (2023). In-Context Retrieval-Augmented Language Models. In *Trans. Assoc. Comput. Linguistics* 11, pp. 1316–1331 (cited on page 3).

Ray, Baishakhi, Vincent J. Hellendoorn, et al. (2016). 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*, pp. 428–439 (cited on page 189).

Raychev, Veselin (2016). Learning from Large Codebases. PhD thesis. ETH Zurich, Zürich, Switzerland (cited on page 84).

Raychev, Veselin, Pavol Bielik, and Martin T. Vechev (2016). Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pp. 731–747 (cited on page 55).

Raychev, Veselin, Martin T. Vechev, and Andreas Krause (2019). Predicting program properties from 'big code'. In *Commun. ACM* 62.3, pp. 99–107 (cited on page 55).

Raychev, Veselin, Martin T. Vechev, and Eran Yahav (2014). 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*, pp. 419–428 (cited on page 55).

Raymond, Eric S. (1998). The Cathedral and the Bazaar. In *First Monday* 3.3 (cited on page 1).

Reid, Machel, Nikolay Savinov, et al. (2024). Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. In *CoRR* abs/2403.05530. arXiv: 2403.05530 (cited on page 222).

Reimers, Nils and Iryna Gurevych (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pp. 3980–3990 (cited on page 46).

Ren, Liliang, Yang Liu, et al. (2024). Samba: Simple Hybrid State Space Models for Efficient Unlimited Context Language Modeling. In *CoRR* abs/2406.07522. arXiv: 2406.07522 (cited on page 222).

Ren, Ruiyang, Yingqi Qu, et al. (2021). RocketQAv2: A Joint Training Method for Dense Passage Retrieval and Passage Re-ranking. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pp. 2825–2835 (cited on pages 29, 138).

Ren, Shuo, Daya Guo, et al. (2020). CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. In *CoRR* abs/2009.10297. arXiv: 2009.10297 (cited on page 133).

Robertson, Stephen E., Steve Walker, et al. (1994). Okapi at TREC-3. In *Proceedings of The Third Text REtrieval Conference, TREC 1994, Gaithersburg, Maryland, USA, November 2-4, 1994*. Vol. 500-225. NIST Special Publication, pp. 109–126 (cited on pages 44 sq.).

Robertson, Stephen E. and Hugo Zaragoza (2009). The Probabilistic Relevance Framework: BM25 and Beyond. In *Found. Trends Inf. Retr.* 3.4, pp. 333–389 (cited on pages 7, 44, 133, 145, 148, 240).

Roy, Chanchal Kumar and James R Cordy (2007). A Survey on Software Clone Detection Research. In *Queen's School of computing TR* 541.115, pp. 64–68 (cited on page 143).

Sadowski, Caitlin, Kathryn T. Stolee, and Sebastian G. Elbaum (2015). How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pp. 191–201 (cited on page 162).

Sahavechaphan, Naiyana and Kajal T. Claypool (2006). XSnippet: Mining For Sample Code. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pp. 413–430 (cited on page 136).

See, Abigail, Peter J. Liu, and Christopher D. Manning (2017). Get To The Point: Summarization with Pointer-Generator Networks. 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*, pp. 1073–1083 (cited on page 55).

Sengamedu, Srinivasan and Hangqi Zhao (2022). Neural language models for code quality identification. In *Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE 2022, Singapore, Singapore, 18 November 2022*, pp. 5–10 (cited on page 190).

Sennrich, Rico, Barry Haddow, and Alexandra Birch (2016). Neural Machine Translation of Rare Words with Subword Units. 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* (cited on pages 16 sq., 55, 65).

Shah, Haseeb, **Johannes Villmow**, and Adrian Ulges (2020). Relation Specific Transformations for Open World Knowledge Graph Completion. In *Proceedings of the Graph-based Methods for Natural Language Processing (TextGraphs)*. Barcelona, Spain (Online), pp. 79–84 (cited on page 10).

Shah, Haseeb*, **Johannes Villmow***, et al. (2019). An Open-World Extension to Knowledge Graph Completion Models. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pp. 3044–3051 (cited on pages 10, 27).

Shanahan, Murray (2024). Talking about Large Language Models. In *Commun. ACM* 67.2, pp. 68–79 (cited on page 78).

Shaw, Peter, Jakob Uszkoreit, and Ashish Vaswani (2018). Self-Attention with Relative Position Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pp. 464–468 (cited on pages 36 sqq., 52, 58 sq.).

Shazeer, Noam (2020). GLU Variants Improve Transformer. In *CoRR* abs/2002.05202. arXiv: 2002.05202 (cited on page 96).

Shazeer, Noam and Mitchell Stern (2018). Adafactor: Adaptive Learning Rates with Sublinear Memory Cost. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Vol. 80. Proceedings of Machine Learning Research, pp. 4603–4611 (cited on page 22).

Shi, Haoyue, Hao Zhou, Jiaze Chen, and Lei Li (2018). On Tree-Based Neural Sentence Modeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pp. 4631–4641 (cited on pages 67, 242).

Shiv, Vighnesh Leonardo and Chris Quirk (2019). Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 12058–12068 (cited on pages 52, 56, 64 sq., 69, 72, 240).

Siegmund, Janet, Norman Peitek, et al. (2017). Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pp. 140–150 (cited on pages 184, 188).

Sim, Susan Elliott, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes (2011). How Well Do Search Engines Support Code Retrieval on the Web? In *ACM Trans. Softw. Eng. Methodol.* 21.1, 4:1–4:25 (cited on page 162).

Sindhgatta, Renuka (2006). Using an information retrieval system to retrieve source code samples. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pp. 905–908 (cited on pages 44, 134).

Singer, Janice, Timothy C. Lethbridge, Norman G. Vinson, and Nicolas Anquetil (1997). An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada*, p. 21 (cited on page 161).

Socher, Richard, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng (2013). Reasoning With Neural Tensor Networks for Knowledge Base Completion. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems*

2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States, pp. 926–934 (cited on page 54).

Sohn, Kihyuk (2016). Improved Deep Metric Learning with Multi-class N-pair Loss Objective. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 1849–1857 (cited on page 29).

Sparck Jones, Karen (1972). A statistical interpretation of term specificity and its application in retrieval. In *Journal of documentation* 28.1, pp. 11–21 (cited on page 44).

Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le (2014). 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*, pp. 3104–3112 (cited on pages 54 sq.).

Svajlenko, Jeffrey and Chanchal K. Roy (2015). Evaluating clone detection tools with BigCloneBench. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pp. 131–140 (cited on pages 128, 133, 143, 237).

Svyatkovskiy, Alexey, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan (2020). IntelliCode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pp. 1433–1443 (cited on pages 84, 127).

Sweller, John (2011). CHAPTER TWO - Cognitive Load Theory. In vol. 55. Psychology of Learning and Motivation, pp. 37–76 (cited on page 188).

Tai, Kai Sheng, Richard Socher, and Christopher D. Manning (2015). Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pp. 1556–1566 (cited on pages 54, 56, 242).

Takang, Armstrong A., Penny A. Grubb, and Robert D. Macredie (1996). The effects of comments and identifier names on program comprehensibility: an experimental investigation. In *J. Program. Lang.* 4.3, pp. 143–167 (cited on pages 187 sq.).

Taylor, Wilson L. (1953). "Cloze Procedure": A New Tool for Measuring Readability. In *Journalism Quarterly* 30.4, pp. 415–433. eprint: https://doi.org/10.1177/107769905303000401 (cited on page 131).

Thies, Andreas and Christian Roth (2010). Recommending rename refactorings. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE 2010, Cape Town, South Africa, May 4, 2010*, pp. 1–5 (cited on page 189).

Tipirneni, Sindhu, Ming Zhu, and Chandan K. Reddy (2024). StructCoder: Structure-Aware Transformer for Code Generation. In *ACM Trans. Knowl. Discov. Data* 18.3, 70:1–70:20 (cited on page 116).

Tu, Zhaopeng, Zhendong Su, and Premkumar T. Devanbu (2014). 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*, pp. 269–280 (cited on page 84).

Tufano, Michele, Cody Watson, et al. (2019). An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In *ACM Trans. Softw. Eng. Methodol.* 28.4, 19:1–19:29 (cited on pages 104, 106 sq.).

Vaswani, Ashish, Noam Shazeer, et al. (2017). Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008 (cited on pages 2, 31, 33–37, 41, 51 sqq., 55, 58, 67 sq., 96, 242).

Villmow, Johannes, Viola Campos, Adrian Ulges, and Ulrich Schwanecke (2022). Addressing Leakage in Self-Supervised Contextualized Code Retrieval. In *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*, pp. 1006–1013 (cited on pages 10, 77, 127, 159, 237, 240, 279).

Villmow, Johannes*, Viola Campos*, et al. (2023b). How Well Can Masked Language Models Spot Identifiers That Violate Naming Guidelines? In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*, pp. 131–142 (cited on pages 10, 77, 183, 186, 242, 279).

Villmow, Johannes, Jonas Depoix, and Adrian Ulges (2021a). ConTest: A Unit Test Completion Benchmark featuring Context. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Online, pp. 17–25 (cited on page 11).

Villmow, Johannes, Adrian Ulges, and Ulrich Schwanecke (2021b). A Structural Transformer with Relative Positions in Trees for Code-to-Sequence Tasks. In *International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021*, pp. 1–10 (cited on pages 9, 51, 57, 69 sq., 73 sq., 225 sq., 241, 279).

Villmow, Johannes, Adrian Ulges, and Ulrich Schwanecke (2024). Evaluating Contextualized Code Search in Practical User Studies. In *INFORMATIK 2024, Wiesbaden, Germany, 24. September – 26. September 2024*. Vol. 352. LNI, pp. 1393–1403. ISBN: 978-3-88579-746-3 (cited on pages 10, 279).

Villmow, Johannes, Marco Wrzalik, and Dirk Krechel (2018). Automatic Keyphrase Extraction Using Recurrent Neural Networks. In *Machine Learning and Data Mining in Pattern Recognition - 14th International Conference, MLDM 2018, New York, NY, USA, July 15-19, 2018, Proceedings, Part II*. Vol. 10935. Lecture Notes in Computer Science, pp. 210–217 (cited on page 10).

Vinyals, Oriol, Meire Fortunato, and Navdeep Jaitly (2015). 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*, pp. 2692–2700 (cited on pages 16, 55).

Viola, Paul A. and Michael J. Jones (2004). Robust Real-Time Face Detection. In *Int. J. Comput. Vis.* 57.2, pp. 137–154 (cited on page 171).

Wang, Alex, Amanpreet Singh, et al. (2019). GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019* (cited on page 78).

Wang, Changhan, Kyunghyun Cho, and Jiatao Gu (2020). Neural Machine Translation with Byte-Level Subwords. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pp. 9154–9160 (cited on page 16).

Wang, Xin, Yasheng Wang, et al. (2021a). CLSEBERT: Contrastive Learning for Syntax Enhanced Code Pre-Trained Model. In *CoRR* abs/2108.04556. arXiv: 2108.04556 (cited on pages 85, 242).

Wang, Yue, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi (2021b). CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pp. 8696–8708 (cited on pages 16, 43, 79 sqq., 86, 92 sq., 103, 114, 209, 240).

Weimer, Westley, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest (2009). Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pp. 364–374 (cited on page 17).

White, Martin, Christopher Vendome, Mario Linares Vásquez, and Denys Poshyvanyk (2015). Toward Deep Learning Software Repositories. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, pp. 334–345 (cited on page 84).

Wilcoxon, Frank (1945). Individual Comparisons by Ranking Methods. In *Biometrics Bulletin* 1.6, pp. 80–83. ISSN: 00994987 (cited on page 175).

Williams, Ronald J. and David Zipser (1989). A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. In *Neural Comput.* 1.2, pp. 270–280 (cited on page 26).

Wrzalik, Marco, Julian Eversheim, et al. (2023). Value Stream Repair Using Graph Structure Learning. In *Advances and Trends in Artificial Intelligence. Theory and Applications - 36th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2023, Shanghai, China, July 19-22, 2023, Proceedings, Part II*. Vol. 13926. Lecture Notes in Computer Science, pp. 15–32 (cited on page 11).

Wrzalik, Marco and Dirk Krechel (2021). CoRT: Complementary Rankings from Transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pp. 4194–4204 (cited on page 29).

Wu, Felix, Angela Fan, et al. (2019). Pay Less Attention with Lightweight and Dynamic Convolutions. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019* (cited on pages 67, 241).

Xia, Congying, Chenwei Zhang, et al. (2019). Multi-grained Named Entity Recognition. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019,*

Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers, pp. 1430–1440 (cited on page 171).

Xu, Xiaojun, Chang Liu, and Dawn Song (2017). SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. In *CoRR* abs/1711.04436. arXiv: 1711.04436 (cited on pages 51, 55).

Ye, Yunwen and Gerhard Fischer (2002). Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pp. 513–523 (cited on pages 134, 136).

Yin, Pengcheng and Graham Neubig (2017). 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*, pp. 440–450 (cited on page 56).

Yu, Hao, Wing Lam, et al. (2019). Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 70–80 (cited on page 51).

Zhang, Dongxu and Dong Wang (2015). Relation Classification via Recurrent Neural Network. In *CoRR* abs/1508.01006. arXiv: 1508.01006 (cited on page 54).

Zhang, Jingqing, Yao Zhao, Mohammad Saleh, and Peter J. Liu (2020). PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research, pp. 11328–11339 (cited on page 83).

Zhao, Wayne Xin, Kun Zhou, et al. (2023). A Survey of Large Language Models. In *CoRR* abs/2303.18223. arXiv: 2303.18223 (cited on page 78).

Zhou, Yaqin, Shangqing Liu, et al. (2019). Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 10197–10207 (cited on pages 51, 104, 108 sq.).

Ziegler, Albert, Eirini Kalliamvakou, et al. (2024). Measuring GitHub Copilot's Impact on Productivity. In *Commun. ACM* 67.3, pp. 54–63 (cited on page 161).

Software

Babelfish, Babelfish 2020. URL: https://github.com/bblfsh, (visited on 07/31/2024) (cited on page 19).

Biewald, Lukas, Experiment Tracking with Weights and Biases 2024. URL: https://www.wandb.com/, (visited on 06/07/2024) (cited on pages 23, 105).

Brunsfeld, Max, tree-sitter version v0.20.0, 2023. URL: https://github.com/tree-sitter/tree-sitter, (visited on 12/27/2023) (cited on pages 17, 98, 245).

Falcon, William and The PyTorch Lightning Team, PyTorch Lightning version v1.4, 2019. DOI: 10.5281/zenodo.3828935, (cited on pages 103 sq.).

OpenAI, ChatGPT version GPT-4o, 2024. URL: https://chatgpt.com, (visited on 05/28/2024) (cited on pages 3, 43, 127, 181).

OpenAI, OpenAI Text Embedding Models version text-embedding-3-{small,large}, 2024. URL: https://platform.openai.com/docs/guides/embeddings, (visited on 09/27/2024) (cited on pages 150 sq., 222).

Ramírez, Sebastián, FastAPI version 0.88, 2023. URL: https://fastapi.tiangolo.com, (visited on 09/01/2024) (cited on page 164).

Stanford NLP Group, Stanford CoreNLP version 3.9.2, 2018. URL: https://stanfordnlp.github.io/CoreNLP, (visited on 10/05/2018) (cited on page 67).

Thunes, Chris, javalang version v0.12.0, 2023. URL: https://github.com/c2nes/javalang, (visited on 12/27/2023) (cited on page 17).

Villmow, Johannes, Tensortree version v0.2.0, 2021. URL: https://github.com/villmow/tensortree, (visited on 09/07/2024) (cited on pages 6, 82, 100 sq., 244).

Villmow, Johannes, Contextualized Code Search Replication Package 2022. URL: https://github.com/villmow/coling-cocos, (visited on 10/04/2024) (cited on page 7).

Villmow, Johannes, CodeBuddy 2024. URL: https://github.com/villmow/codebuddy, (visited on 10/04/2024) (cited on page 8).

Villmow, Johannes, Viola Campos, et al., CodeDoctor Replication Package 2023. DOI: 10.5281/zenodo.7612762, (cited on pages 9, 187).

You, Evan, Vue.js - The Progressive JavaScript Framework version 3.2.45, 2024. URL: https://vuejs.org/, (visited on 09/01/2024) (cited on page 164).

Zayarni, André, Qdrant - High-performance, massive-scale Vector Database version v1.6.1, 2023. URL: https://github.com/qdrant/qdrant, (visited on 05/07/2024) (cited on pages 27, 165).

Web References

Bitkom (2024). Mangel an IT-Fachkräften droht sich dramatisch zu verschärfen. de. URL: https://www.bitkom.org/Presse/Presseinformation/Mangel-an-IT-Fachkraeften-droht-sich-zu-verschaerfen (visited on 10/04/2024) (cited on page 1).

Gage, Philip (1994). FEB94 A New Algorithm for Data Compression. URL: http://www.pennelynn.com/Documents/CUJ/HTML/94HTML/19940045.HTM (visited on 08/16/2024) (cited on page 16).

GitClear, Harding William, and Matthew Kloster (2024). Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality. URL: https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality (visited on 10/15/2024) (cited on pages 158, 221).

GitHub (2024). GitHub Copilot · Your AI pair programmer. URL: https://github.com/features/copilot (visited on 05/28/2024) (cited on pages 3, 42 sq., 127, 161, 181, 184).

GitHub Inc. (2024). GitHub Activity Data – Marketplace – Google Cloud Console. URL: https://console.cloud.google.com/marketplace/details/github/github-repos (visited on 04/28/2024) (cited on page 102).

Lopes, C., S. Bajracharya, J. Ossher, and P. Baldi (2010). UCI Source Code Data Sets. URL: https://ics.uci.edu/~lopes/datasets/index.html (visited on 01/24/2024) (cited on page 66).

Munroe, Randall (2020). XKCD Comic #2309: X. URL: https://xkcd.com/2309/ (visited on 08/28/2024) (cited on page 183).

NetBSD (2024). NetBSD Commit Guidelines. URL: https://www.netbsd.org/developers/commit-guidelines.html (visited on 05/18/2024) (cited on page 128).

Oracle (2024). How to Write Doc Comments for the Javadoc Tool. URL: https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html (visited on 02/01/2024) (cited on page 66).

Salva, Ryan J. (2023). Introducing Code Referencing for GitHub Copilot. URL: https://github.blog/2023-08-03-introducing-code-referencing-for-github-copilot/ (visited on 05/28/2024) (cited on pages 128, 161, 181).

Tabnine (2024). Tabnine AI code assistant. en-US. URL: https://www.tabnine.com/ (visited on 10/16/2024) (cited on page 43).

The Standish Group (2013). Chaos Manifesto: Think Big, Act Small. URL: https://larlet.fr/static/david/stream/ChaosManifesto2013.pdf (visited on 10/04/2024) (cited on pages 1 sq.).

Tools

List of tools used for this dissertation:

- DeepL (https://www.deepl.com/)
- DeepL Write (https://www.deepl.com/en/write)
- ChatGPT (https://chat.openai.com/)
- Grammarly (https://www.grammarly.com/)
- Zotero (https://www.zotero.org/)
- DBLP (https://dblp.org/)
- Python
- Matplotlib
- TikZ (PGFPlots)
- LaTeX

Own Contributions

The following list provides a list of my contributions to the work presented in the chapters of this dissertation:

Chapter 3 This work was completely done by me and first presented in *"A Structural Transformer with Relative Positions in Trees for Code-to-Sequence Tasks"* (**Villmow**, Ulges, and Schwanecke 2021b). Adrian Ulges provided guidance and together with Ulrich Schwanecke proofread the document.

Chapter 4 This work was completely done by me and not yet published.

Chapter 5 The work to this chapter was first presented in *"Addressing Leakage in Self-Supervised Contextualized Code Retrieval"* (**Villmow**, Campos, Ulges, and Schwanecke 2022). Compared to the original paper, I added additional details, examples, figures, and experiments, hoping to provide a more comprehensive demonstration of the approach. This research project was a collaborative effort primarily led by me. I want to greatly thank Viola Campos for her help in annotating half of the dataset and implementing the BM25 baseline. I wrote all other code, also collected the dataset, and ran the experiments. In the publication, co-author Viola Campos contributed to the related work section and proofread the document, and Adrian Ulges also provided proofreading

Chapter 6 This work was completely done by me and first presented in *"Evaluating Contextualized Code Search in Practical User Studies"* (**Villmow**, Ulges, and Schwanecke 2024). Adrian Ulges proofread the paper.

Chapter 7 This work was first presented in "How Well Can Masked Language Models Spot Identifiers That Violate Naming Guidelines?" (**Villmow***, Campos*, Petry, Andaloussi, Ulges, and Weber 2023b). This research project was a collaboration with the *Software Systems Programming and Development* group at the *University of St.Gallen*. Many thanks to my co-authors Viola Campos, Jean Petry, Amine Abbad-Andaloussi, Adrian Ulges, and Barbara Weber for their valuable contributions to this work. Amine Abbad-Andaloussi and Barbara Weber provided guidance on the cognitive aspects of source code, including the identification of naming guidelines. Also, both contributed to the methodology of the experiments (particularly, the selection of suitable guidelines) and to the related work about the comprehension of source

code. The dataset was annotated by a team of students from the *RheinMain University of Applied Sciences*. I had the idea to estimate the likelihood of an identifier using an Language Model (LM), trained the generative LM, and developed and implemented the initial framework for the approach for detecting, masking, and scoring variables. This includes the perplexity-based scoring functions and the evaluation procedure for the dataset. Viola Campos had the idea to the likelihood-ratio scoring function, and added the InCoder baseline to the framework. Jean Petry implemented the discriminative scoring function advised by Adrian Ulges and me. All authors contributed to the manuscript, and the final version of the manuscript was reviewed by all authors. Compared to the work published in this paper, I added a new scoring function, extended the description of approach and dataset, and discussed the results in more detail.

Erklärung

Hiermit erkläre ich, Johannes Villmow, dass ich die vorliegende Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) mit dem Titel

Self-Supervised Learning on Source Code to Assist Software Developers

selbständig, ohne fremde Hilfe und nur mit den angegebenen Hilfsmitteln angefertigt habe. Alle Textstellen, die wörtlich oder sinngemäß aus veröffentlichten Werken übernommen wurden, sowie alle Aussagen, die auf mündlichen Informationen beruhen, sind als solche gekennzeichnet. Die Grundsätze guter wissenschaftlicher Praxis wurden beachtet.

Wiesbaden, 23. Oktober 2024

Johannes Villmow

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Adobe Garamond, a revival of Claude Garamont's humanist typeface. A template that can be used to format a PhD dissertation with this look & feel has been released under the permissive AGPL license, and can be found online at github.com/suchow/Dissertate or from its lead author, Jordan Suchow, at suchow@post.harvard.edu.