Using NLP techniques for automated code refactoring

IME USP

Alan Barzilay

Using Natural Language Processing techniques for automated code refactoring

Alan Barzilay

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Master of Science

Program: Computer Science
Advisor: Prof. Dr. Marcelo Finger

This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil* (CAPES) – Finance Code 001. This study was also financed in part by the grant #2020/02679-4 from the São Paulo Research Foundation (FAPESP).

São Paulo
March, 2023

Using Natural Language Processing techniques for automated code refactoring

Alan Barzilay

This version of the thesis includes the corrections and modifications suggested by the Examining Committee during the defense of the original version of the work, which took place on March 27, 2023.

A copy of the original version is available at the Institute of Mathematics and Statistics of the University of São Paulo.

Examining Committee:

Prof. Dr. Marcelo Finger (advisor) – IME-USP
Prof. Dr. Eduardo Martins Guerra – unibz
Prof. Dr. Aline Marins Paes Carvalho – UFF

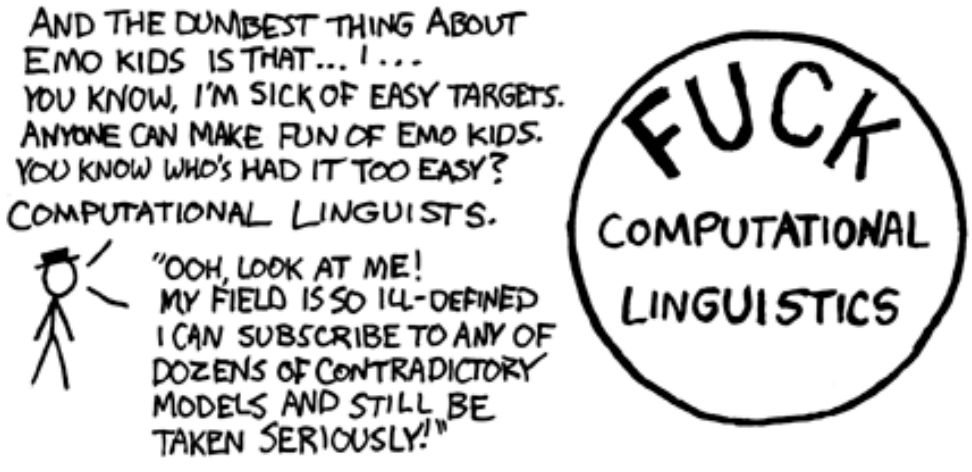
Ficha catalográfica elaborada com dados inseridos pelo(a) autor(a)
Biblioteca Carlos Benjamin de Lyra
Instituto de Matemática e Estatística
Universidade de São Paulo

Barzilay, Alan
Utilizando técnicas de processamento de linguagem natural para refatoração automática de código / Alan Barzilay; orientador, Marcelo Finger. - São Paulo, 2023.
115 p.: il.

Dissertação (Mestrado) - Programa de Pós-Graduação em Ciência da Computação / Instituto de Matemática e Estatística / Universidade de São Paulo.
Bibliografia
Versão original

1. Aprendizado de Máquina. 2. Refatoração. 3. Processamento de Linguagem Natural. 4. Engenharia de Software. I. Finger, Marcelo. II. Título.

Bibliotecárias do Serviço de Informação e Biblioteca
Carlos Benjamin de Lyra do IME-USP, responsáveis pela
estrutura de catalogação da publicação de acordo com a AACR2:
Maria Lúcia Ribeiro CRB-8/2766; Stela do Nascimento Madruga CRB 8/7534.



Randall Munroe (2006)

Agradecimentos

Gostaria de agradecer à minha família pelo apoio e sustento que me proporcionaram ao longo dessa longa jornada como mestrando. Especialmente meu primo, David Barzilai, que me acompanhou enquanto eu escrevia parte dessa tese e até me abrigou em sua casa por uns dias enquanto eu terminava mais uma versão do texto.

Ainda sobre abrigo, gostaria de agradecer meu amigo Lucas Magno por sempre me receber de portas abertas em sua sala. A vida acadêmica pode ser bem solitária, mas sempre ter com quem bandejar certamente ajuda.

Gostaria também de agradecer meus amigos Mauricio, Zeca e a família deles[1] por sempre me receberem e me alimentarem quando eu ia passar o dia trabalhando lá. Não sei o por que eles achavam tão razoável e normal eu ir lá com tanta frequência mas sou muito grato por isso. Além de me receberem, eles auxiliaram essa tese de diversas formas diretas e indiretas; por exemplo, o Zeca, a Lira e a Júlia Rissin me resgataram de carro uma madrugada quando fiquei ilhado no meio do nada voltando da casa deles. E ainda quando o Mauricio resgatou todos os dados perdidos do HD quando o disco `sdb` da máquina ratel faleceu com o "tec-tec da morte".

Falando na ratel, gostaria de agradecer ao Sergio Ricardo Milare e ao William Alexandre Miura Gnann da seção de informática do IME pela ajuda com essa máquina e em recuperar a última versão do meu código que eu tinha esquecido de dar `git push`. Tecnicamente a ratel nem era responsabilidade deles então sou muito grato por sempre serem tão solícitos e não só ignorarem nossos emails falando que eu quebrei a ratel de novo.

Gostaria de aproveitar para também agradecer a equipe dos restaurantes universitários, equipes de limpeza da USP, responsáveis da sala do chá do IME, a secretaria da pós do IME, o Renato Geh, Felipe "Sub" Serras e demais RDs e representantes do corpo estudantil, o Nelson Lago, o professor Carlinhos (que me fez desistir das minhas ideias ruins e só entrar no mestrado do IME logo) e todos os outros funcionários e professores do IME e da USP

[1] Menos a gata Gigi

que me proporcionaram as condições de desenvolver esse projeto.

Não posso deixar de agradecer meu querido orientador Marcelo Finger que sempre me deu muita liberdade e apoio ao longo do projeto, sempre trazendo novas ideias e oportunidades para mim.

Gostaria de agradecer a Júlia Rissin também pela ajuda com os gráficos, me salvando de apresentar uma tese cheia de *printscreens*.

Gostaria também de agradecer todos os meus amigos e colegas que fizeram parte dessa minha jornada do mestrado, como o Gabriel Morete, Lucas Arenstein, Thiago Lima, Rebecca Helena, Pedro Souza (cara mais fascinado por `git` que eu já vi), Thiago Tarraf Varella, Estêvão, Pedro Bruel, Rodrigo Berezovsky, Arieh Szafir Goldstein, Matheus Castello, Roger Bravo, e meu colegas do LIAMF, por conta da pandemia nossa convivência foi curta mas foi um prazer conhecer todos vocês.

Por fim gostaria de agradecer meu grande amigo Patrick Eli Catach. Por conta da distância entre a Inglaterra e o Brasil as oportunidades de nos vermos são raras, mas isso nunca o impediu de se interessar pelo meu trabalho — às vezes até mais do que eu — e de sempre ser o primeiro a querer ler meus rascunhos incompletos. Seu apoio foi muito importante para mim e para esse projeto, seja me recebendo nas suas raras visitas ao Brasil ou quando ele veio me visitar na Holanda e passamos boa parte do tempo discutindo meu projeto, silenciosamente trabalhando lado a lado ou até mesmo vindo me cobrar de mandar minha tese para ele ler. Para ele, a maioria desses acontecimentos devem ter sido corriqueiros e sem muito significado, mas para um amigo que mora tão longe e que poderia ter facilmente acabado se afastando e perdendo o contato ao sair do país — como aconteceu com tantos outros no meio desse êxodo acadêmico — sua dedicação e sinceridade foram extremamente marcantes.

I would also like to thank Minaksie Ramsoekh for all the help in getting to Delft and during my stay there.

They say science is built on-top of the shoulders of giants, however due to an archaic publishing model and corporate greed most of the population has no access to these giants to even attempt to climb on-top of them. That is why I would like to thank everyone involved in the dissemination of free and open science and in particular to Alexandra Elbakyan and her project Sci-Hub which was essential in the realization of this thesis.

Resumo

Alan Barzilay. **Utilizando técnicas de processamento de linguagem natural para refatoração automática de código**. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Técnicas de processamento de linguagem natural podem ser aplicadas aos mais diversos textos, não somente àqueles redigidos em linguagens humanas como também àqueles redigidos em linguagens ditas artificiais, como códigos escritos em linguagens de programação. Refatoração de código é uma técnica fundamental em engenharia de software, sendo utilizada tanto como uma ferramenta para garantir a qualidade do código como também como um passo importante na expansão de funcionalidades e depuração. Nesse trabalho propomos um modelo para refatoração automática de código. Ao utilizar diretamente o código fonte como entrada em nosso modelo de processamento de código nós obtemos automaticamente sugestões de refatorações do tipo extração de função. O modelo proposto consiste em uma rede neural capaz de receber uma representação vetorial do código a ser refatorado e gerar uma representação da refatoração sugerida. Essa rede foi treinada com base numa lista de repositórios de código obtida através de uma colaboração com a TU Delft na Holanda. Com base nessa lista foi criado o maior dataset de refatorações de extração de função existente — até o momento da publicação dessa tese — sendo 60% maior do que o segundo maior dataset de seu tipo. Além disso, nosso modelo final atingiu uma acurácia de teste de 0.7275.

Palavras-chave: Aprendizado de Máquina. Refatoração. Processamento de Linguagem Natural. Engenharia de Software.

Abstract

Alan Barzilay. **Using Natural Language Processing techniques for automated code refactoring**. Thesis (Master's). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Natural Language Processing techniques can be applied to text in general, not only to human language but also to artificial languages such as software code. Code refactoring is a fundamental software engineering technique used both as a quality assurance tool and an important step in code correction and functionality enhancement. In this work, we propose a novel code refactoring model. By utilizing source code as input to our model we obtain automated suggestions of function extraction code refactoring in order to achieve better readability and attain good practices in general. The proposed model consists of a neural network that receives a vectorial representation of the source code and outputs a representation of the suggested refactored code. This network was trained based on a list of repositories provided through a collaboration with TU Delft Holland. Based on this list we created the biggest existing function extraction refactoring dataset — as of the time this thesis was presented — being %60 bigger than the second biggest dataset of its type. Furthermore, our final model achieved a test accuracy of 0.7275.

Keywords: Machine Learning. Code Refactoring. Natural Language Processing. Software Engineering.

List of Abbreviations

NLP	Natural Language Processing
ML	Machine Learning
LSP	Language Server Protocol
AST	Abstract Syntax Tree
DFA	Discrete Finite Automata
GloVe	Global Vectors (for Word Representation)
SQuAD	Stanford Question Answering Dataset
BERT	Bidirectional Encoder Representations from Transformers
SBERT	Sentence-BERT
RoBERTa	Robustly optimized BERT approach
ALBERT	A Lite BERT
QA	Question Answering
dbmc1	distiluse-base-multilingual-cased-v1
Ptr-Net	Pointer Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
TPE	Tree-structured Parzen Estimator
Acc	Accuracy
Val	Validation
IDE	Integrated Development Environment
ORM	Object Relational Mapping
FAPESP	São Paulo Research Foundation (Fundação de Amparo à Pesquisa do Estado de São Paulo)
BEPE	Grant for Research Abroad (Bolsa Estágio de Pesquisa no Exterior)
TUDelft	Delft University of Technology (Technische Universiteit Delft)
SERG	Software Engineering Research Group

List of Figures

List of Tables

List of Programs

Contents

Chapter 1

Introduction

What is the difference between computer science and software engineering? Both deal with computers, right? Are they not the same thing?

Computer science is the field focused on the theoretical powers of computers and the algorithms we write for them. Is this problem decidable? NP-complete? Can this thing be considered a Turing machine? Is P=NP? What functions can be approximated by neural networks? How can we color, with the minimal number of colors, the vertices of a graph so that no two adjacent vertices have the same color? There are a myriad of important and interesting questions posed in this large field of study, but they are all centered around the computers and the theoretical.

This is where the computer science and software engineering fields differentiate themselves. Whereas one field is centered around computers and algorithms, the other is centered around the humans using these computers and implementing those algorithms, or to be more precise the relation and interactions between programmers and their code.

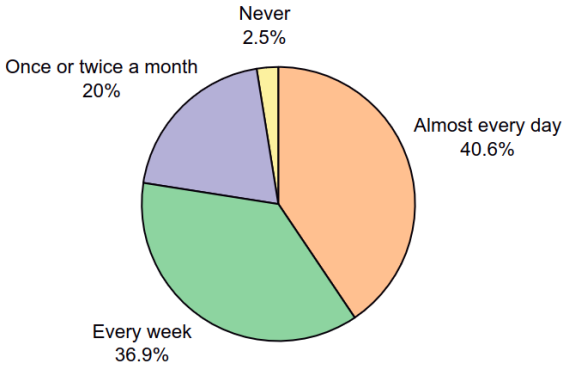
Software engineering is the field concerned with more mundane and practical aspects of programming, when we speak of code complexity we are talking about a more abstract concept than the well defined time complexity so ubiquitous in computer science theory, we are talking about the *perceived* complexity of a program[1]. How easy is it to maintain? Is it readable? Can we easily test it for limit cases where bugs may be hiding? Is there code duplication that could be simplified?

Software engineering is the field of study concerned with the efficiency of the programmers and their code, not simply their code. How can we decrease the implementation time of a new feature and make developers more efficient? What are the bottlenecks in development? How do the developers organize themselves? What if they are part of a team? What about their code base? How do we measure the quality of a piece of code in regards to the value it brings to our clients or even to our own developers?

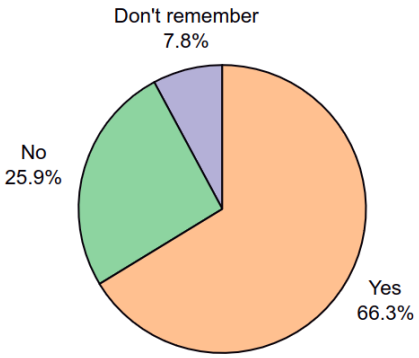
When trying to add a new feature to an existing software, a common first step is the

[1] This is not to say that software engineers have no interest in or knowledge of the theoretical aspects of computer science or that the two fields are completely disjoint, the best developers would take into account both the code's time complexity and its perceived complexity when writing a program.

re-writting of existing code, not the creation of new code as one might expect (Fowler, 1999). The field of software engineering has a special name for this re-writting step, it is called *refactoring*. To refactor a piece of code is to make simple and incremental changes in order to make it easier to work with and to be expanded upon. A survey from Golubev *et al.* (2021) shows that refactorings are a key element in the software development cycle. As can be seen in Figs. 1.1, nearly four out of five developers in the 1,183 developers surveyed indicated that they refactored code every week or even almost every day. Two-thirds of these respondents said that they had refactoring sessions of an hour or longer during this time.



(a) *In the past month, how often have you performed any code refactoring? (Out of 1,181 respondents)*



(b) *During this time, did you ever refactor code for an hour or more in a single session? (Out of 1,145 respondents)*

Figure 1.1: *Pie graphs of two of the 20 questions about refactoring answered by a group of 1,183 developers that were paying subscribers of IntelliJ Platform based IDEs. (Golubev et al., 2021)*

As one might expect from such an integral part of the software development cycle, extensive work (Golubev *et al.*, 2021; Kim *et al.*, 2014; Abid *et al.*, 2020) has been done to better understand how and when refactorings are done, or how to optimize and automate this often manual and repetitive task. However, much of the work done is in order to automate a refactoring after a developer detected an opportunity and it is mediated by said developer. Let's say we want to break a long function into smaller functions, the developer

has to go to the function and select the line span that they wish to extract. The onus of refactoring the code is still on the developer, it is certainly faster than doing it manually without the help of these small automations but there is still room for improvement. Taking the same example as before, it would be even better if the IDE could suggest to the programmer which functions could use a refactoring or even which lines should be extracted. We posit that by making the refactoring process more automated and less reliant on the developer, refactoring would become even more commonplace and make developers in general more productive.

But how exactly could we achieve such automation? That is what we hope to answer in detail in the following chapters of this thesis dissertation. We intend to leverage the *naturalness* of programming languages to train a machine learning model based on existing natural language processing techniques to automate refactorings of the function extraction type.

One might ask what is the "naturalness of a programming language" and why should we use NLP techniques in languages that are, by their very definition, non-natural but first we need to address what even is a natural language.

The concept of natural language could be defined by what they are not: they are not artificially constructed and they are not "rigid". Let us take Hope C. Dawson (2016) definition of natural languages:

> *All languages exhibit all nine design features* **[as defined by C. F. Hockett and C. D. Hockett (1960): Mode of Communication, Semanticity, Pragmatic Function, Interchangeability, Cultural Transmission, Arbitrariness, Discreteness, Displacement, Productivity]** *any communication system that does not is therefore not a language. Furthermore, as far as we know, only human communication systems display all nine design features.* **[...]**
>
> *Because all languages exhibit the nine design features, does this mean that any communication system that exhibits all nine features should be considered a language? For example, there are formal languages, such as the formal logic used to write mathematical proofs and various computer languages. While these formal languages display all of the design features, they nevertheless differ in critical ways from languages such as English, Spanish, Mandarin, and Apache. For example, no child could ever acquire a computer language like C++ as his native language! Furthermore, a number of people engage in constructing languages that imitate human language as a hobby. There are many reasons that people might choose to do this. For example, the created language could be used in some sort of fictional universe, such as Klingon in the television series Star Trek or Dothraki and Valyrian in the series Game of Thrones. Or it might be designed to facilitate international communication, which was the goal of the designers of the language Esperanto. Other people, such as J.R.R. Tolkien, have constructed artificial languages just for fun.*
>
> *Do we want to make a distinction between languages such as English, Spanish, Mandarin, and Apache, on the one hand, and Esperanto, Elvish, Dothraki, Valyrian, and Klingon, on the other? And how should we classify*

> *'formal' languages? Although many of these questions are still open to debate and research, we will make the following distinctions* **[...]** *we call natural languages, those languages that have evolved naturally in a speech community. The lexicon and grammar of a natural language have developed through generations of native speakers of that language. A constructed language, on the other hand, is one that has been specifically invented by a human and that may or may not imitate all the properties of a natural language.*
> Hope C. Dawson (2016)

Essentially, natural languages are languages that changed and evolved naturally alongside humans and are used for communication. That is not to say that a constructed language cannot become a natural language, an example is Modern Hebrew which was reconstructed and expanded from Ancient Hebrew (a liturgical and dead language) and then adopted by a particular community[2]. We used non rigid to describe natural languages because of this capacity to change and evolve as well as its semantical robustness, even if a text contains a few mistakes its meaning may still be grasped while programming languages are brittle to mistakes (e.g. typos in variable names or parameter order inversion can drastically change the meaning of code or even break it).

Hope C. Dawson (2016) also brings us a sort of definition for formal languages:

> *The distinction between constructed languages and formal languages is that formal languages are not the sort of system that a child can acquire naturally.*
> Hope C. Dawson (2016)

Although this is not wrong, most computer scientists would find it lacking as a definition. A more common formulation would be to define it as the set of all strings that can be derived from a formal grammar. That is to say, a formal language is defined (or generated) by its grammar and a piece of text can be identified as part of a formal language if it can be "recognized" by its grammar (i.e. parsed). More formally, a generative grammar is defined by a 4-tuple (N, Σ, P, S) where N is the set of non-terminal symbols, Σ the set of terminal symbols (disjoint of N), P the set of production or generation rules and S the sentence or start symbol.

Table 1.1 depicts an example of the production rules part of a formal grammar for syntactically correct infix algebraic expressions for three variables, namely x, y and z, and depicts Fig 1.2 an example of a parse tree generated with these production rules for the expression $(x + y) \times x - z \times y/(x + x)$.

Formal languages, such as programming languages, can still change and evolve, however this happens as punctuated changes (e.g. the release of Python 3 to substitute Python 2 and the sub-sequential break of compatibility) and in contrast to natural languages that work in a bottom up fashion through social dynamics (Croft, 2008), programming languages are designed top-down by a few designers for many users.

These are not their only differences, another example is that it is not clear if it is feasible

[2] The matter of Modern Hebrew's "creation", "revival" or "(Re)vernacularization"(Spolsky, 1995) and its characterization as a constructed language is a point of contention, this discussion is out of the scope of this work but we refer readers interested in a primer on the subject to Izre'el (2003).

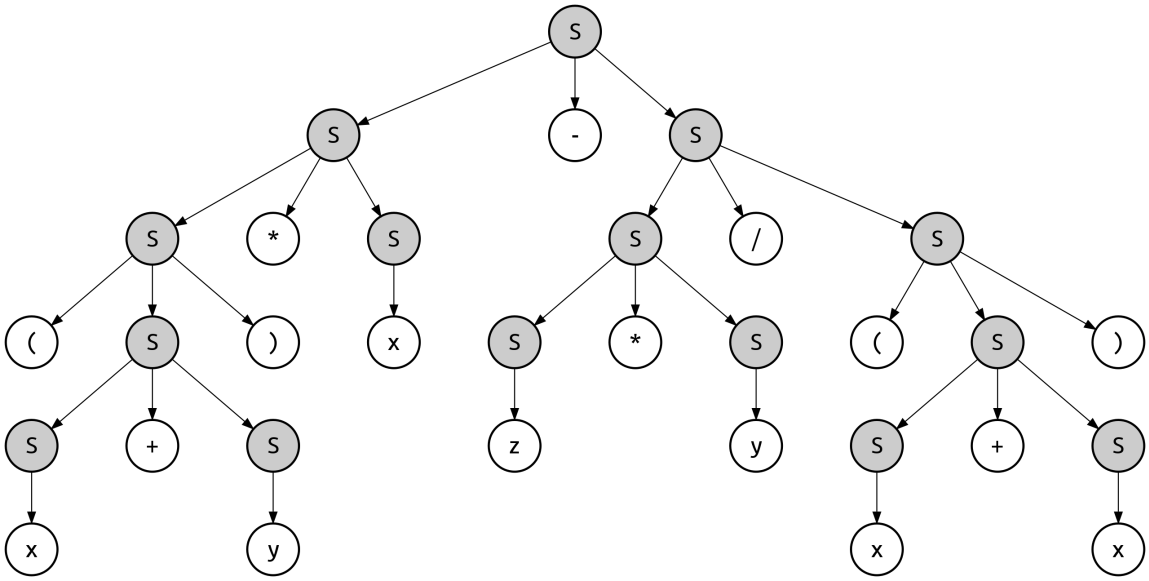


Figure 1.2: *One of the possible parse trees for the expression* $(x + y) \times x - z \times y/(x + x)$ *created from the production rules in Table 1.1. Image from* Pat Hawks *(2018).*

$S \to x$	$S \to S + S$
$S \to y$	$S \to S - S$
$S \to z$	$S \to S \times S$
$S \to (S)$	$S \to S/S$

Table 1.1: *Production rules of a formal grammar for syntactically correct infix algebraic expressions for three variables, namely x, y and z. In essence, the grammar is defined by these 8 production rules forming the P set,* $\{x, y, z, +, -, (,), \times, /\}$ *as the terminal symbols set* Σ, $\{S\}$ *as the set of non-terminal symbols N and S as the start symbol. Note that this grammar is ambiguous so it has multiple possible parse trees.*

to create a translation between natural languages such that the meaning is completely preserved[3]. However every mainstream[4] programming language is Turing complete (Sipser, 2013) so it is always possible[5] to exactly translate a piece of code from one language to another.

Nevertheless machine translation between natural languages is an ever thriving research topic. Furthermore, in recent years the area of NLP has seen explosive growth together with new neural network architectures (GRU (Cho *et al.*, 2014), Transformers (Vaswani *et al.*, 2017), ELMo (Peters *et al.*, 2018), BERT (Devlin *et al.*, 2018) to name only a few recent ones) and a skyrocketing success reaching even mainstream popularity

[3] And also highly dependent on the definition of *meaning* being used.

[4] There are programming languages that are not developed with universal computation in mind, such as the BlooP (Hofstadter, 1979) and Charity (The Charity Development Group, 1996) languages, however they are not widely adopted in the industry nor posses a sizable user base.

[5] Anyone who has worked in code translation may tell you that this can be "easier said than done". Code portability can be a challenging subject even if we ignore unusual Turing complete languages such as Microsoft's PowerPoint (Wildenhain, 2017).

through large language models such as GPT-4 (OpenAI, 2023) and LLaMA (Touvron *et al.*, 2023). On the other hand, the use and success of machine learning models for analogous tasks in programming languages (e.g. translation between programming languages or transpilation) has been comparatively moderate so far.

Despite having several differences, natural and programming languages also share a series of particularly interesting commonalities that allow the application of models and techniques originally devised for NLP tasks into analogous programming language processing tasks.

This idea is succinctly defined in the naturalness hypothesis:

> **Naturalness Hypothesis.** *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.*
> Allamanis, Earl T Barr, *et al.* (2018)

This insight that programming may be seen as a form of human communication is not by any means new and can be traced back to Donald E. Knuth concept of *literate programming* from his titular work Literate Programming:

> *I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: 'Literate Programming.'*
>
> *Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*
>
> *The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reïnforce each other.*
> Knuth (1984)

Although the naturalness hypothesis may not seem surprising to some, it is worth understanding the genesis of this naturalness.

As stated by Allamanis, Earl T Barr, *et al.* (2018) "*naturalness of code seems to have a strong connection with the fact that developers prefer to write (Allamanis, Earl T. Barr, et al., 2014) and read (Hellendoorn et al., 2015) code that is conventional, idiomatic, and familiar because it helps understanding and maintaining software systems*".

This leads to the idea that code artifacts may contain recurring and predictable patterns that can be leveraged by machine learning models to perform a plethora of different tasks,

in a not dissimilar way to how recurring and predictable patterns in linguistic corpora have been successfully utilized in NLP models.

Hindle *et al.* (2016) demonstrated that corpus-based statistical language models can capture a high level of regularity in software, even more so than in english, and that this is not an artifact of the programming language syntax but rather it arises from the naturalness of the code.

They reason that, like natural languages, software is repetitive and predictable:

> *We begin with the conjecture that* most software is also natural*, in the sense that it is created by humans at work, with all the attendant constraints and limitations* **[...]**
>
> *Programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks.*
> Hindle *et al.* (2016)

An important point raised was that **most** software is also natural, not all code ever written. One could easily argue that the hypothesis of human communication can be discarded when we are dealing with esoteric programming languages such as brainfuck (Urban Müller, 1993), Piet (Thomas Schoch, 2006a) or Malbolge (Ben Olmstead, 1998), aptly named after the 8th circle of Hell from Alighieri (130-). Fig. 1.3 presents three code snippets to help clarify how convoluted and deliberately obtuse esoteric languages can be designed to be. Being less extreme, not all code could be said to be "readable" or to be clearly and well structured, e.g. an inexperienced programmer's code[6], may not conform to best practices and contain confusing or out-write miss-leading function and variable names. If this is sufficient to classify the code as not natural is a matter out of the scope of this project, however we posit that those are rare instances in large successful projects that usually have naming conventions, contributing guidelines and code quality metrics and as such would not have a huge impact on data quality and training performance.

So far, this naturalness approach to programming languages has been shown to be fruitful many times. Allamanis, Earl T Barr, *et al.* (2018) make an extensive review of the literature compiling works that leverage in some way the ideas behind the naturalness hypothesis, there are too many to list here so we encourage interested readers to seek the original paper that is readily available online. Readers pressed for time may be interested in inspecting the tables since they compile most of the surveyed work in a systematic and well organized manner. It is important to note that this is not an exhaustive survey

[6] Recent evidence actually suggests that the amateur nature of inexperienced programmers may actually be beneficial for the corpus as was described in testimony given by Replit's Head of AI in Latent Space (2023), where they claim to have obtained a 50% increase in performance when fine tuning their model on the replit codebase — which contains a lot of code from people still in the process of learning how to program. However, our point stands that even if not all code could be considered natural it is likely relegated to a small part of softwares at large and would end up as an issue of data quality the same way data quality is an issue for NLP corpora.

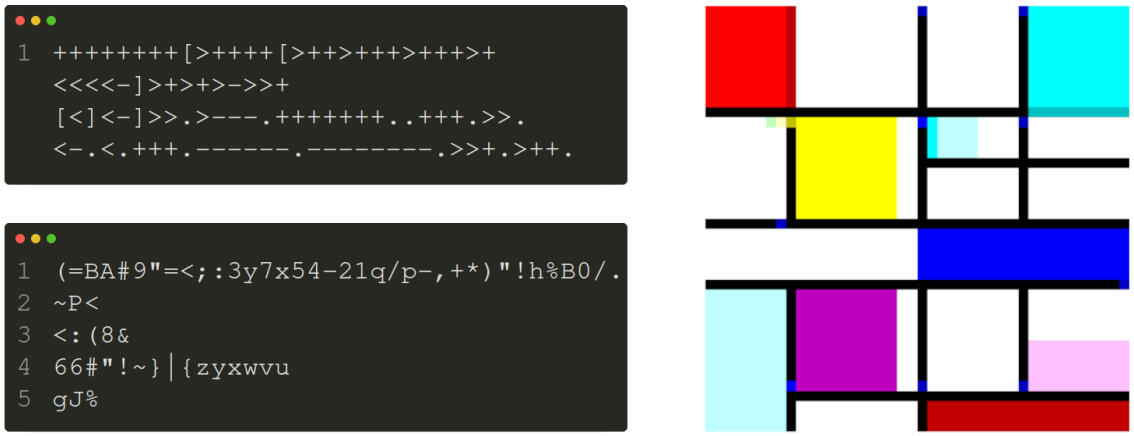


Figure 1.3: *3 code snippets in esoteric languages. On the right the image that resembles a painting from Piet Mondrian is a program in the Piet language that prints the "Piet" string (Thomas Schoch, 2006b). On the upper left is a "Hello World!" program written in brainfuck where only 8 characters (><+-.,[]) are available and each correspond to a different pointer operation (Esolang wiki, 2023a). Lastly, on the lower left we have a* `cat` *program (that does not stop at EOF) written in Malbolge, a language designed to be as difficult to program in as possible with a ternary system, self-altering code and, once again, only 8 valid instructions (Esolang wiki, 2023b).*

since it was published in 2018 and this field of study has by no means stopped since then. Two interesting and more recent works are the code2vec (Alon, Zilberstein, *et al.*, 2018), capable of abstracting the work done by a function by naming it based solely on information obtained through its abstract syntax tree and code2seq (Alon, Brody, *et al.*, 2018), not only capable of predicting method names but also predicting natural language captions given partial and short code snippets, and to even generate method documentation.

We believe these ideas and the results we obtained during the course of this project serve as a proof of the soundness of our approach.

1.1 Goals

This project intends to accomplish two main goals:

- Understand if deep learning models are capable of predicting fine-grained refactorings, i.e. where exactly the source code should be refactored.
- Create a model for automated function extraction.

1.2 Organization

The thesis is organized as follows: in Chapter 2 we define refactorings and propose a theoretical IDE plugin to better illustrate our objectives as well as provide some software engineering background; in Chapter 3 we present the theoretical background of our models and other NLP and ML concepts; in Chapter 4 we present related work on automating refactorings; in Chapter 5 we explain how we built our dataset; in Chapter 6

we define the building blocks of our model followed by our experiments with those blocks in Chapter 7; and, finally, in Chapter 8 we give our concluding remarks and trace possible future steps.

Chapter 2

An imaginary function extraction plugin

Although every modern IDE has a series of tools for automating common simple code refactorings, the automation of suggestions of refactorings is still lacking. E.g., renaming a variable can be easily accomplished by tools that scan your program for instances of this variable in the appropriate scope. But there is no efficient tool to our knowledge that automatically detects the need to rename a variable and suggests a new name.

In this chapter we will propose the structure of how one could create an IDE plugin that automates refactorings of the function extraction type and suggests instances of such refactorings. Our objective is by no means the construction nor the implementation of this imaginary plugin, as previously stated in our goals we only intend to create a model that, given only a function definition, is capable of performing a function extraction. We propose this plugin as a mental exercise to clarify some of the practical uses of the work proposed in this project and as a way to introduce software engineering concepts that the readers may need to fully comprehend to understand this project.

2.1 Code Refactoring

This project aims at automating refactorings but we never actually defined what refactoring is, so let us look at some definitions starting with the one who coined the term:

> *This thesis defines a set of program restructuring operations (refactorings) that support the design, evolution and reuse of object-oriented application frameworks. [...] The refactorings are defined to be behavior preserving, provided that their preconditions are met. Most of the refactorings are simple to implement and it is almost trivial to show that they are behavior preserving.* Opdyke (1992)

Another simpler definition from another pioneer on the subject, Martin Fowler, could be useful to those new to the term:

> **Refactoring** (noun): *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*
> Fowler (1999)

> **Refactoring** (verb): *to restructure software by applying a series of refactorings without changing its observable behavior.*
> Fowler (1999)

From these definitions we can gather that ideally each refactoring would be a simple and small step that a programmer could take in order to improve the perceived complexity of a program and be more compliant with best practices. After taking a series of these small steps a program will have an improved maintainability, being more readable and easier to debug while behaving the same exact way. Fig. 2.1 shows an example of such an operation, a refactoring categorized as *Combine Functions into Class*.

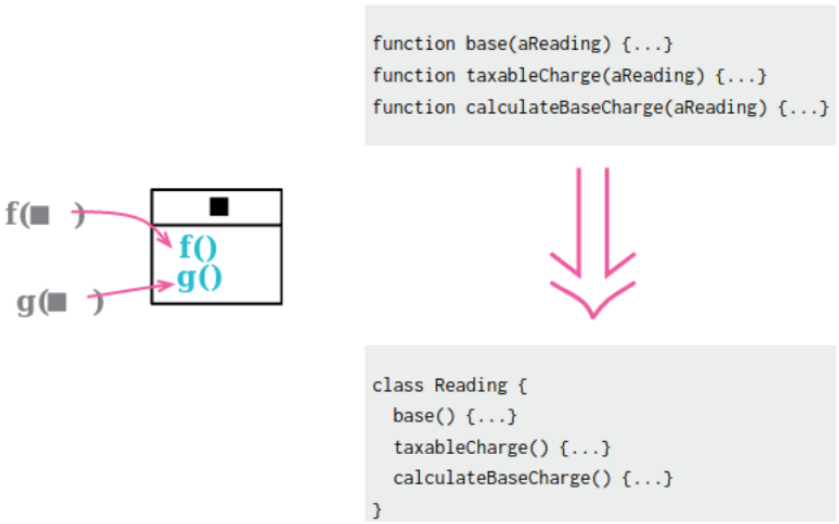


Figure 2.1: *An instance of the "Combine Functions into Class" refactoring. Example extracted from (Fowler, 1999).*

However, in practice this definition of refactoring being a behavior-preserving code transformation does not always hold true. It is not uncommon to use the term refactoring in ways that defy its academic definition, Martin Fowler addresses this:

> *Over the years, many people in the industry have taken to use 'refactoring' to mean any kind of code cleanup—but the definitions above point to a particular approach to cleaning up code. Refactoring is all about applying small behavior-preserving steps and making a big change by stringing together a sequence of these behavior-preserving steps. Each individual refactoring is either pretty small itself or a combination of small steps. As a result, when I'm refactoring, my code doesn't spend much time in a broken state, allowing me to stop at any moment even if I haven't finished.*

> *If someone says their code was broken for a couple of days while they are refactoring, you can be pretty sure they were not refactoring.*
> Fowler (1999)

That is to say, colloquially refactoring is often used as something more generic and less rigorous than Opdyke's or Fowler's definitions. A survey of 328 professional software engineers at Microsoft found that developers do not necessarily consider that refactoring is confined to behavior preserving transformations (Kim *et al.*, 2014). Furthermore:

> *[...] 78% define refactoring as code transformation that improves some aspects of program behavior such as readability, maintainability, or performance. 46% of developers did not mention preservation of behavior, semantics, or functionality in their refactoring definition at all. [...] The following shows a few examples of refactoring definitions by developers.*
> *'Rewriting code to make it better in some way.'*
> *'Changing code to make it easier to maintain. Strictly speaking, refactoring means that behavior does not change, but realistically speaking, it usually is done while adding features or fixing bugs.'*
> Kim *et al.* (2014)

Bearing in mind those different meanings, we will stick to Martin Fowler's definition unless stated otherwise. Martin Fowler (2023a) has a catalog where he defines many different types of refactorings but we are interested in automating a specific type of refactoring, the function extraction. When our hypothetical plugin suggests a function extraction to its fictional user it will only suggest the extraction, no feature will be added or subtracted. Further code improvements are out of the scope of this project and are left as possible paths for future work.

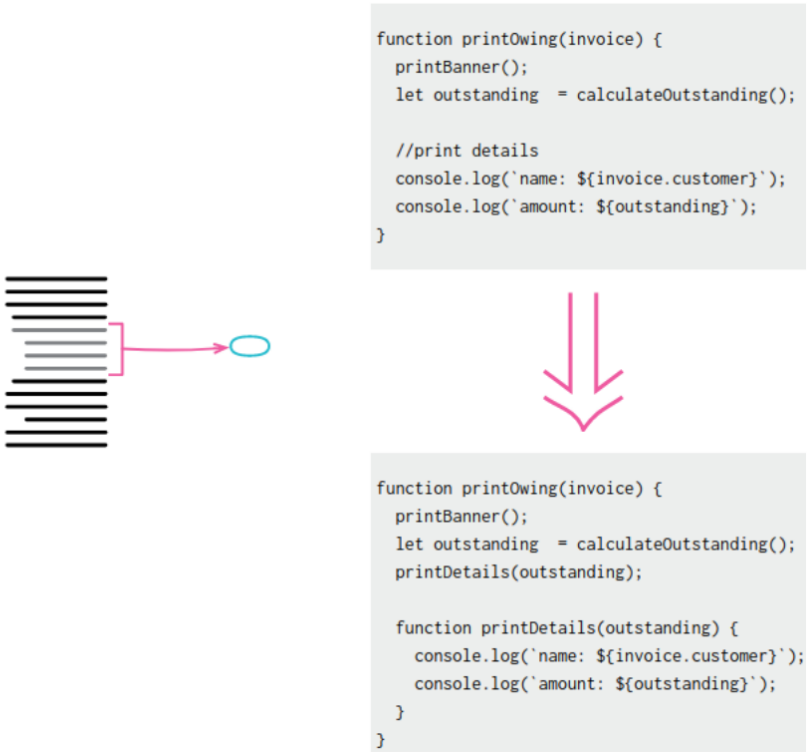
Following Martin Fowler's nomenclature system (Fowler, 1999), in this project we will explore the *function extraction* refactoring in particular, where a long function that does many tasks is broken down into smaller functions, that only do one simple task each, being called one after the other. Each new function created from the larger original one is an instance of a function extraction. It is not uncommon to have a series of function extractions being applied sequentially to a single large function. Fig. 2.2 gives an example of this refactoring.

2.2 The Plugin

With a clear definition of what is a refactoring and of our objectives we will explore our theoretical plugin and how it could be built. Fig. 2.3 presents the outline of how this plugin would work.

2.2.1 Detecting Refactoring Opportunities

As previously stated, code refactoring is not a new subject, extensive work has already been done to classify different types of refactorings. Furthermore, there are no lack of tools (Palomba *et al.*, 2013; Tufano *et al.*, 2015; Tsantalis, Chaikalis, *et al.*, 2008; Daniel *et al.*, 2007) to identify *code smells* (Fowler, 1999), a common pattern that may serve



```
function printOwing(invoice) {
  printBanner();
  let outstanding  = calculateOutstanding();

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
}
```

```
function printOwing(invoice) {
  printBanner();
  let outstanding  = calculateOutstanding();
  printDetails(outstanding);

  function printDetails(outstanding) {
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
  }
}
```

Figure 2.2: *An instance of the "Function Extraction" refactoring. Example extracted from (Fowler, 1999).*

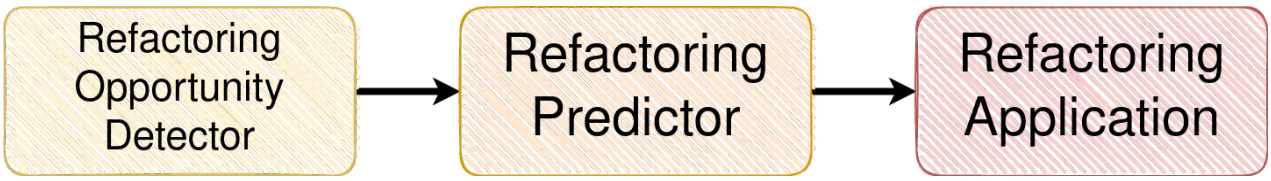


Figure 2.3: *Outline of our imaginary plugin broken down into three different parts.*

as an indication of a deeper problem in the way the system is structured and the need of refactoring. For example a really long function may indicate the need for a function extraction refactoring.

There are many approaches one could take in order to identify refactoring opportunities. One approach could be to leverage said code smells and their vast literature, another way could be to construct heuristics utilizing a set of code complexity metrics.

There are quite a few metrics that aim at measuring the perceived complexity of a piece of code — not in the sense of *execution time complexity* but in the sense of *readability and maintainability*. By looking solely at the source code, without executing it, it is possible to provide an estimation of the quality of the software. Some metrics can be more specific and restrictive in their use, for example being tailored to a single programming paradigm (e.g. this suite of metrics designed for Object Oriented programming (Chidamber and Kemerer, 1994)). Other metrics may try to identify "code smells" or be specific to a single programming language (gojp, 2023).

Which of these ideas is the best we cannot say without testing and the results would ultimately be dependent on the implementation of the heuristics, but since we are dealing with an imaginary plugin we are not concerned with such details. We will explore the simple code complexity metric known as *maintainability index* in order to better illustrate this piece of our imaginary plugin.

In section 4.3 we will see another possible approach that utilizes machine learning to detect refactoring opportunities and the refactoring type to be used.

Cyclomatic complexity

The cyclomatic complexity proposed in 1976 (McCabe, 1976), may be defined as the number of linearly independent paths within a piece of code. A program with no control flow statements (such as loops and conditionals) will have a cyclomatic complexity of 1, a program with one conditional statement if will have cyclomatic complexity of 2, one independent path for a True if statement and another for a False if statement. Cyclomatic complexity may be used as a bound for the number of necessary unit tests for 100% code coverage and a high cyclomatic complexity may be an indicative of complex nested flow statements.

Halstead Metrics

The Halstead metrics where developed in 1977 (Maurice, 1977) as an attempt to define and analyze static properties of a given software, trying to capture a measure of how difficult it is to write or understand a given piece of code. They also estimate how many bugs are present in a given snippet of code. Let,

η_1 = Number of distinct operators

η_2 = Number of distinct operands

N_1 = Total number of operators

N_2 = Total number of operands

Halstead defines volume, difficulty, effort and estimated number of bugs as:

$$V = (N_1 + N_2) * \lg(\eta_1 + \eta_2)$$

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

$$E = D * V$$

$$\hat{B} = \frac{E^{\frac{2}{3}}}{3000}$$

Maintainability Index

Throughout the years, the maintainability index formula has been tweaked and refined, but on the original paper (Coleman *et al.*, 1994) it was defined as:

$$MI = 171 - 5.2 * \ln(V) - 0.23 * G - 16.2 * \ln(LoC)$$

Where V is the Halstead Volume, G is the Cyclomatic Complexity and LoC is the total number of lines of code. Typical values for the maintainability index range from 0 to 100 with 0 being a hard to maintain code and 100 being a well structured, small and easy to maintain code.

2.2.2 Refactoring Prediction

This is the kernel of this plugin and also our main objective in this project. As such, let us treat it as a black box model that simply works until we arrive at Chapter 6 where we will present our attempts at solving this problem.

2.2.3 Performing Refactorings

Language Server Protocol (Microsoft, 2016), or LSP as it is commonly referred, was introduced to address the duplication of effort and code across IDE's and text editors. The idea is to create a standard that allows a plugin for a given language (e.g. auto-completion for python) to work in any development tool (e.g. a plugin designed for Atom will work in VSCode) thus reducing the workload of language providers and tooling vendors. Fig 2.4 illustrates this idea.

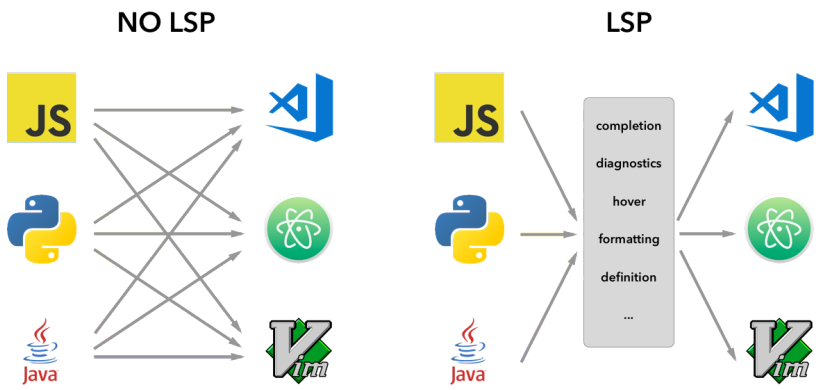


Figure 2.4: *An illustration of the problem that LSP was designed to solve. Without LSP we have M languages that need to have support implemented in N different IDE's, but with LSP's we only need to implement support for a language once and it can be re-used anywhere (Microsoft, 2021).*

The goal of LSP is to allow the implementation and distribution of support for a programming language without involving any particular text editor by standardizing the interaction between the IDE's and the servers that provide language specific features.

With the use of a language server the function extraction refactoring could be easily accomplished by providing the line span to be extracted, in our particular case we could use the Eclipse JDT Language Server (Eclipse Foundation, 2021) to potentially accomplish this for the Java programming language. By describing a refactoring in terms of operations done through the LSP the instructions generated by our model can be easily utilized in any development environment capable of communicating with a language server. It is important to note that function extractions are not always performed with a continuous set of lines which could prove itself a limitation for currently implemented LSP instructions, e.g. the Eclipse JDT Language Server expects a continuous line span. To perform more

complex function extractions there may be a need to expand current LSP instruction sets.

Chapter 3

NLP Techniques

In this chapter we intend to present an overview of some of the NLP concepts, mostly based on neural networks, which are relevant to comprehend this project. As previously mentioned, we intend to leverage NLP techniques to perform the code processing technique of automating function extractions. While some methods can be used as is, others can barely be used at all. In further chapters their usage and limits for our use case will be further explored, while in this chapter they are only going to be introduced to the readers to better situate them.

3.1 Embeddings

When we are dealing with neural networks, we are essentially dealing with matrix operations, so in order to apply NNs to NLP tasks one first needs to somehow transform words and sentences into numbers.

A rudimentary approach would be an one-hot-encoding, where given a vocabulary of size V we have a vector of size V with each word of the vocabulary being associated to a specific index of our vector in a one-to-one fashion, where it is 0 at every index except for the one corresponding to the word to be represented. Fig. 3.1 illustrates, with categorical variables, the one-hot-encoding. Another similar concept is called bag of words, a sentence representation obtained by adding the one-hot-encoding of the words in said sentence.

However, these approaches are inefficient. Vocabularies tend to be huge, Merriam-Webster's Third New International Dictionary (Merriam-Webster, Inc., 2023) possesses $470,000$ different entries, but a typical sentence would not be longer than 100 words. This leads to a sparse representation of sentences and also loses important information such as the order of appearance of the words in the sentence.

Embeddings are an efficient approach to solve these problems; an embedding is a low-dimensional space (in comparison to the huge size of typical vocabularies) into which one can translate high-dimensional vectors. In essence, an embedding tries to project into a lower dimensional space high-dimensional vectors in such a way that the ones that are "similar" are closer in the embedding space. In the case of words, we are interested in the

Sprite	Name	HP	Attack	Defense	Sp_Atk	Sp_Def	Speed	Type
	Koffing	40	65	95	60	45	35	Poison
	Pikachu	35	55	40	50	50	90	Electric
	Shellder	30	65	100	45	25	40	Water
	Krabby	30	105	90	25	25	50	Water
	Voltorb	40	30	50	55	55	100	Electric
	Cubone	50	50	95	40	50	35	Ground
	Magikarp	20	10	55	15	20	80	Water
	Pineco	50	65	90	35	35	15	Bug
	Misdreavus	60	60	60	85	85	85	Ghost
	Phanpy	90	60	60	40	40	40	Ground

(a) *Categorical representation of Pokémon types.*

Sprite	Name	HP	Attack	Defense	Sp_Atk	Sp_Def	Speed	Poison	Electric	Water	Ground	Bug	Ghost
	Koffing	40	65	95	60	45	35	1	0	0	0	0	0
	Pikachu	35	55	40	50	50	90	0	1	0	0	0	0
	Shellder	30	65	100	45	25	40	0	0	1	0	0	0
	Krabby	30	105	90	25	25	50	0	0	1	0	0	0
	Voltorb	40	30	50	55	55	100	0	1	0	0	0	0
	Cubone	50	50	95	40	50	35	0	0	0	1	0	0
	Magikarp	20	10	55	15	20	80	0	0	1	0	0	0
	Pineco	50	65	90	35	35	15	0	0	0	0	1	0
	Misdreavus	60	60	60	85	85	85	0	0	0	0	0	1
	Phanpy	90	60	60	40	40	40	0	0	0	1	0	0

(b) *One-hot encoding of Pokémon types.*

Figure 3.1: *Table of a few Pokémon statistics representing its type as a **(a)** categorical variable or as an **(b)** one-hot encoding. In this case, the Pokémon types of a trainer could be represented by a bag-of-words by adding the one-hot encodings*[a]*. Base stats from Generation VI from* Bulbapedia *(2005) and images from* Juan Orozco Villalobos *(2020).*

[a] Albeit it would be more of a "bag-of-types" than a bag-of-words

semantic similarity between them but they may be similar in unexpected ways, depending on the training method antonyms may be closer than synonyms because they are used in similar ways.

Embeddings are not exclusive to natural language processing, they are used to deal with the sparsity of adjacency graphs, in graph neural networks, to permit the use of graphs as input for more conventional NNs (Grover and Leskovec, 2016) and may even be used with any machine learning model that deals with too many features, being particularly useful with boolean and categorical variables. They are also not exclusively used with individual words, they may be used with entire sentences (Reimers and Gurevych, 2019), paragraphs or even entire documents at once (Le and Mikolov, 2014).

Another benefit of embeddings is that they have high re-usability, it is not uncommon to re-utilize them across different models and tasks. To further improve their performance there are many transfer learning techniques that can be leveraged, but depending on the application they may not even need fine tuning to achieve an acceptable performance.

We will now present an overview of an embedding called *word2vec* to better illustrate embeddings and their uses.

3.1.1 Word2vec

Word2vec (Mikolov *et al.*, 2013) is an algorithm that produces word embeddings. A word embedding as previously mentioned is a mapping of words into a vector space $\mathbb{R}^n$, i.e. each word can be represented as a vector of real numbers. The power of these embeddings created by word2vec comes from its dense distributed representation, wherein $\mathbb{R}^n$ is such that n is smaller than the number of distinct words in the corpus and is capable of capturing some of the underlying syntactic and semantic structure of the language.

As an example of this structure we can take the vector representation for the words *king*, *queen*, *man* and *woman* and calculate $king - man + woman$. This will provide us a new vector that is closer to *queen* then any other word vector, as can be seen in the illustration of Fig. 3.2. In Figure 3.3 we can see other examples of such relationships captured by word2vec.

These relationships are obtained during its training process, at first each word is assigned a word embedding composed of purely random numbers that will converge into a meaningful embedding. The training process of this algorithm is based on the idea that one may deduce the meaning of a word by its context, so by updating the embedding of a word by using the embedding of its neighbors or vice-versa it is possible to capture semantic and syntactic information from the language thus construing a meaningful embedding space.

This idea of understating a word by its context is nothing new going at least as far as 1957:

> *You shall know a word by the company it keeps.*
> Firth (1957)

Word2vec was one of the first models to efficiently utilize this idea, being quickly

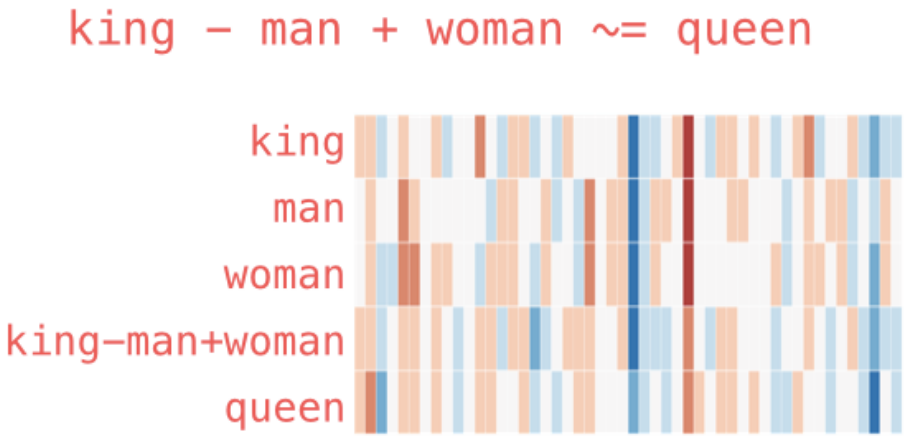


Figure 3.2: *The resulting vector from "king-man+woman" doesn't exactly equal "queen", but "queen" is the closest word to it from the 400,000 word embeddings in this collection. Color coded cells based on their values (red if they're close to 2, white if they're close to 0, blue if they're close to -2). (Jay Alammar, 2019)*

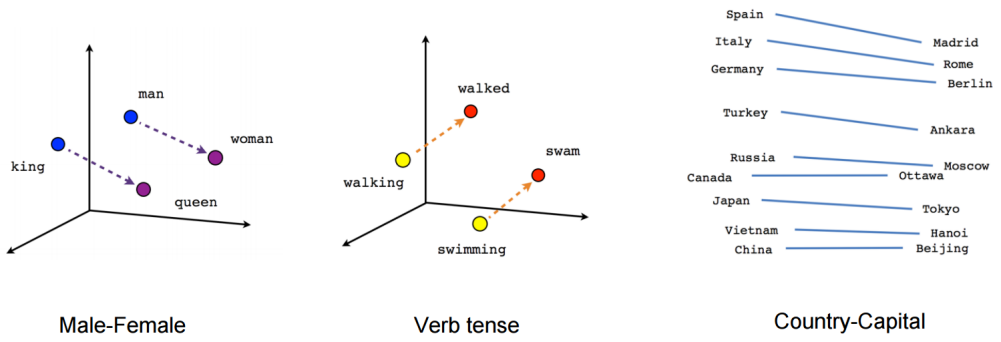


Figure 3.3: *Illustration of some semantic and syntactic relations captured by word2vec embeddings, vectors whose words have similar relationships (such as gender or conjugation) tend to also have similar relations on the vector space (Tensor Flow, 2020). This is nothing more than an illustration since usual embeddings are of such a high order that visualization as a simple 3D plot becomes impossible, dimensionality reduction techniques (e.g. PCA) albeit useful may lead to spurious relations, when searching for such relations it is customary to use appropriate distance metrics such as the cosine distance.*

followed by GloVe (Pennington *et al.*, 2014), a model that leverages the co-ocurrence matrix of words to train its embedding, and many others. An example of a co-ocurrence matrix can be seen in Fig. 3.4. Even though nowadays there are many embeddings more powerful than those two, they are still useful for simpler applications as tried and tested methods or even because of their relatively high computational efficiency.

3.2 LSTM

LSTM, or Long Short-Term Memory (Hochreiter and Schmidhuber, 1997), is a recurrent neural network architecture developed to tackle the problem of vanishing gradient when dealing with long inputs in RNNs where the "gradient flow" would vanish, resulting

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Figure 3.4: *Example of a co-ocurrence matrix with a symmetrical window of size 1.*
Corpus: I like deep learning. I like NLP. I enjoy flying.
Dictionary: ['I', 'like', 'enjoy', 'deep', 'learning', 'NLP', 'flying', '.'] (Christopher Manning, 2020)

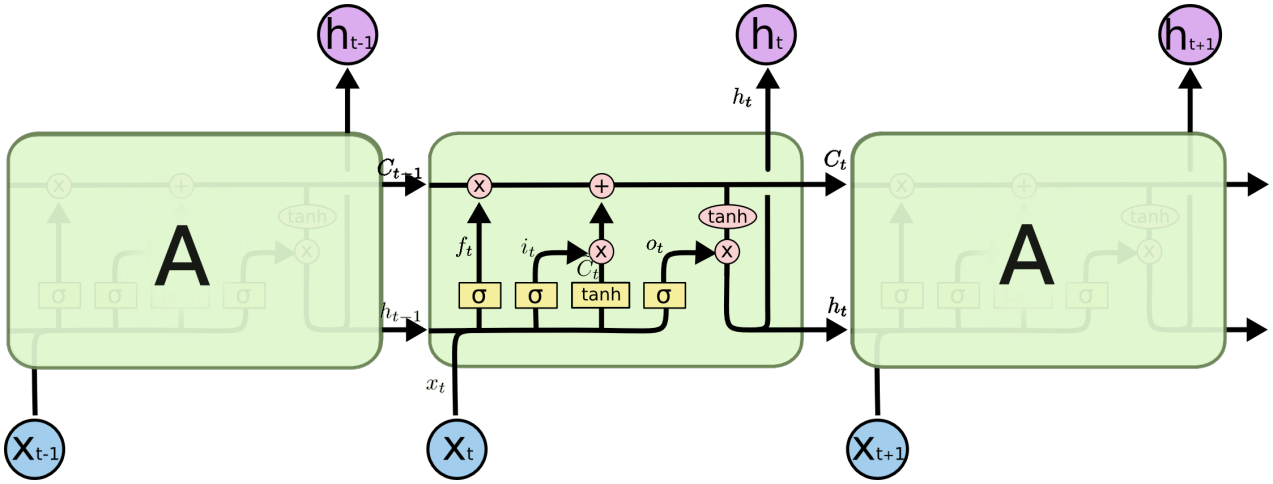


Figure 3.5: *An illustration of the LSTM architecture, image adapted from Christopher Olah (2015).*

in stagnation during training and the inability to deal with long distance relationships in, for example, long sentences or paragraphs. This inability to go beyond "short-term memory", was improved by introducing a better information flow in the network, giving it the ability to learn when to "forget" and when to "update" accordingly. An illustration of this architecture can be seen in Fig. 3.5, but it can be described by its three gates:

Forget gate: $f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$

Input gate: $i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$

Output gate: $o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$

The forget gate is responsible for "deciding" what to keep from earlier steps, the input gate for what to keep from the current step and the output gate for the next hidden state h_t. Another integral part of this model is the cell state C_t and the *candidate* cell state $\tilde{C}_t$, responsible for "storing" the information on the network, being updated at each time step based on the candidate cell state, the input gate and the forget gate:

$$\tilde{C}_t = tanh(W_C.[h_{t-1}, x_t] + b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$h_t = o_t * tanh(C_t)$$

3.3 Encoder-Decoder

The idea behind the encoder-decoder neural network architecture is to use an encoding layer to create an intermediary representation of the input that will subsequently be transformed by the decoder layer into the desired target. Recurrent neural networks such as LSTMs or GRUs (Cho *et al.*, 2014) are commonly used as the encoding and decoding layers to avoid the vanishing gradient problem. A common use of the encoder-decoder architecture is machine translation through seq2seq (Sutskever *et al.*, 2014). For example, an english text can be used as an input to the encoder which will generate a latent space vector, an abstract representation of the text, that will be used by the decoder to generate a french translation of the original text. This process can be visualized in Fig. 3.6.

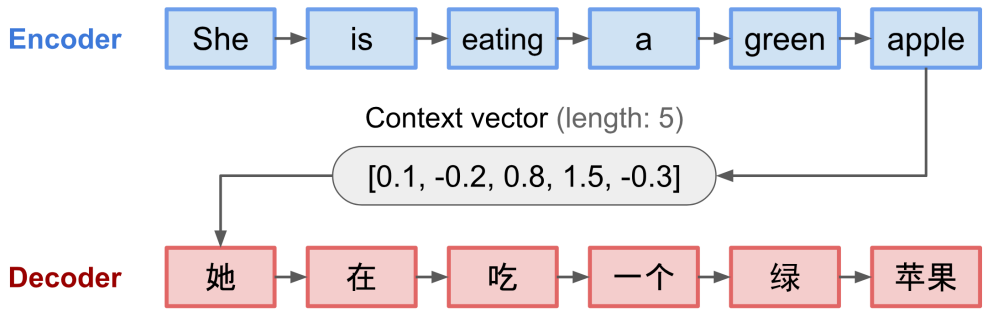


Figure 3.6: *The encoder-decoder model, translating the sentence "she is eating a green apple" to Chinese. The visualization of both encoder and decoder is unrolled in time. (Weng, 2018)*
The context vector in gray is the intermediary representation between the encoder and decoder that needs to hold all the relevant information to successfully realize the translation without referencing the original input in english.

Another example of the encoder-decoder architecture are the autoencoders, where the objective is to make the output predict the input. The existence of an intermediate state outputted by the encoder forces the autoencoder to learn a compact representation of the input and try to reconstruct it as the output, an illustration can be seen in Fig. 3.7.

3.4 Attention

Attention comes to address one of the main drawbacks of the encoder-decoder architectures, the informational bottleneck. By forcing the entire sequence of information to be contained in a single vector it becomes increasingly difficult to capture long-range relations and information presented at the beginning of long input sentences. Attention provides a simple manner of capturing the relevant input tokens for each token of the

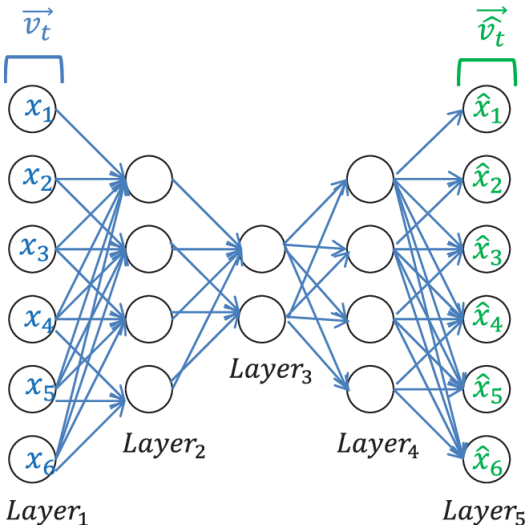


Figure 3.7: *Illustration (Unger et al., 2018) of a shallow autoencoder. $\vec{v}$ and $\vec{\hat{v}}$ represent the input and output respectively of the network while the output from layer 3 is responsible for the compact representation of the input on the latent space. The encoder is composed of layers 1, 2 and 3 while the decoder is composed of layers 4 and 5.*

output, generating an attention matrix that represents the relevant context for each token and improving interpretability.

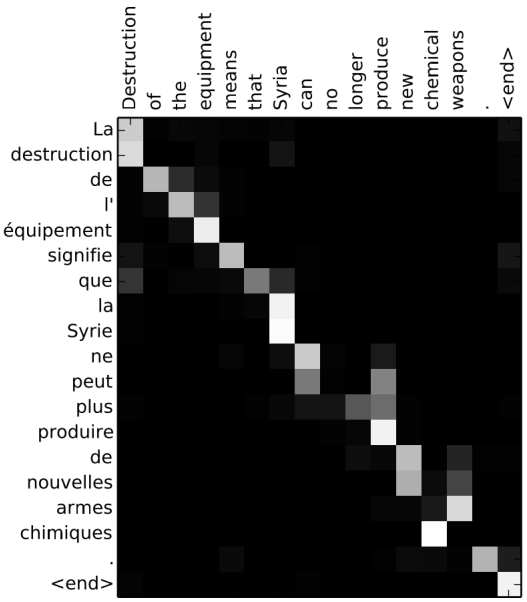


Figure 3.8: *Illustration (Bahdanau et al., 2014b) of an example attention matrix for a sentence in english and it's french translation.*

This provides another counter-measure against the vanishing gradient problem by connecting the entirety of the input to the training process, but the biggest insight of attention is this idea of paying attention to a specific part of the input that is more relevant to obtain our desired output. For example, given the obscured photo in Fig. 3.9a how would one describe its content? We could guess about the gender, origins or facial expression of

the depicted person, but we would most likely assume it depicts a human and not a dog as can be seen in Fig. 3.9b, but how did we arrive at this conclusion? Which part of the image implies the presence of a human? This is the essence of the idea behind attention, where to look at to answer something.

(a) **(b)**

Figure 3.9: *A collage to illustrate the idea of attention and visual attention. Photo obtained from* MENSWEARDOG *(2023).*

GALASSI *et al.* (2021) generalized attention by breaking it into three swappable components: the 3-tuple (key, query, value), the score function and the distribution function.

The tuple (key, query, value) are the input for the score function, the objects upon which we desire to calculate the attention. The score function is used to calculate the energy scores that will be subsequently passed through the distribution function to produce the attention scores such as the *softmax* function that also normalizes the scores into a probability distribution.

Fig. 3.10 illustrates this general framework and Fig. 3.11 gives an illustrative example of one of the first attention implementations.

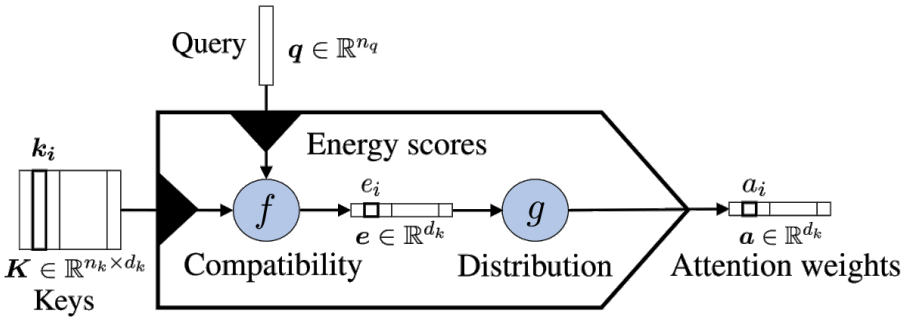
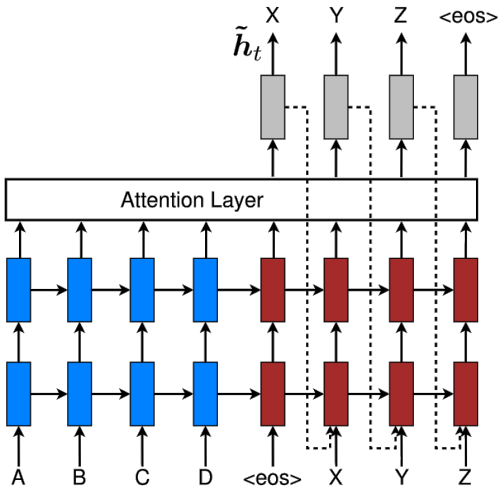
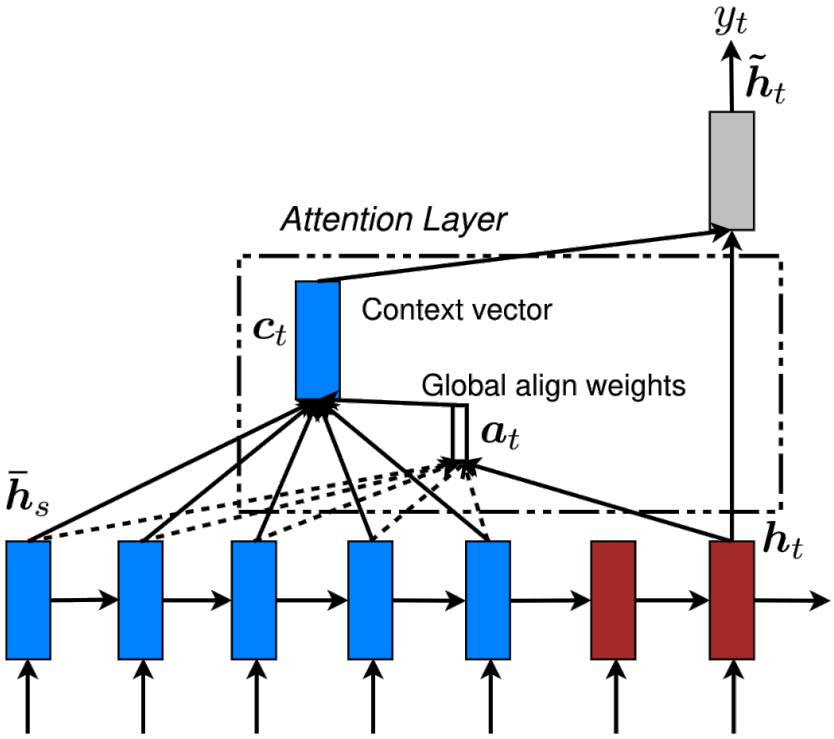


Figure 3.10: *Illustration of* GALASSI *et al. (2021) general attention framework. The "Value" component is not present in this illustration but could be used in a subsequent step to process the attention scores to create a context vector.*



(a) *Illustration of an encoder decoder model with attention, every decoder step decides the next token based on the context vector and the last decoder hidden state. The blue rectangles represent the encoder and the red ones the decoder.*



(b) *To calculate the attention all encoder hidden states $\bar{h}_s$ (Keys)[a] and the current decoder hidden state h_t (Query) are passed to the score function $score(h_t, \bar{h}_s) = h_t^\top \bar{h}_s$ to calculate the energy scores. Then the energy scores are passed to the distribution function softmax $a_t(s) = \frac{exp(score(h_t, \bar{h}_s))}{\sum_{s'} exp(score(h_t, \bar{h}_{s'}))}$ to generate the attention score. How to use an attention score may vary between architectures and applications, in this particular case* Luong et al. (2015) *takes the weighted average $c_t = \frac{\sum_s \bar{h}_i a_i}{|s|}$ of all attention scores to create a context vector which will be used with the decoder hidden state to predict the next word.*

[a] In this case Key and Value are the same

Figure 3.11: *Fig. 3.11a illustrates an encoder decoder model and Fig. 3.11b illustrate its attention mechanism, both images extracted from* Luong et al. (2015).

3.5 Pointer Networks

Pointer Net (Vinyals *et al.*, 2015), or Ptr-Net, is a neural architecture that learns the conditional probability of an output sequence composed of index positions of the input sequence. It is essentially an encoder-decoder architecture with attention, but the attention mechanism is used at each step to determine an index of the input. Fig 3.12 illustrates the difference from a normal encoder-decoder.

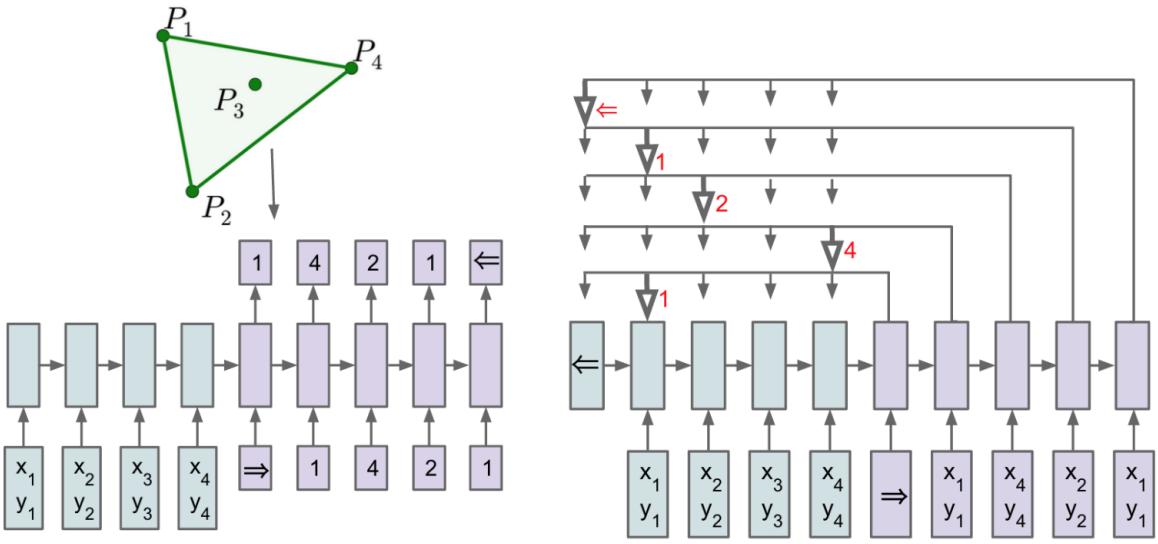


Figure 3.12: *This illustration represents 2 different architectures used to find the convex hull of a set of points (Vinyals et al., 2015). The left one is a normal encoder-decoder while on the right we have a Ptr-Net.*

The original paper explores the use of this architecture with geometric problems, such as finding the convex hull as can be seen in Fig 3.12, but it can also be used for other problems. Another use of Ptr-Nets are in Q&A tasks such as in the SQuAD dataset (Rajpurkar, Zhang, *et al.*, 2016) where a question is provided paired with a text that contains the answer to this question. A common approach (S. Wang and Jiang, 2016) to answer the question is to train a model that finds in the provided text an initial and a final token that together delimit a minimal portion of the text that contains the answer. An example of such a Q&A task can be seen on Fig. 3.13.

3.6 Transformer

The transformer (Vaswani *et al.*, 2017) is another encoder-decoder architecture but with a major difference: it shifts from the use of recurrent neural networks in favor of attention. More precisely, the encoder uses word vectors as input and is composed of a self-attention layer followed by a feed forward network. Self-attention is capable of capturing strong semantic information, for example in the sentence "Marie bought a cookie and ate it." the self-attention model is capable of determining that the token "it" refers to the token "cookie".

The transformer starts by using self-attention on the word embeddings to aggregate

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

What causes precipitation to fall?
gravity

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?
graupel

Where do water droplets collide with ice crystals to form precipitation?
within a cloud

Figure 3.13: *This excerpt illustrates 3 different questions for a single paragraph of text, for the first two questions the initial and final tokens overlap as the answer is composed of a single word. On the third question the initial token would be the word "within" and the final token would be "cloud" and the resulting answer delimited by them would be "within a cloud". Example extracted from (Rajpurkar, Zhang, et al., 2016).*

information from each token, creating a new context-rich representation for each word simultaneously. The decoder on the other hand takes an iterative approach, instead of outputting the entire translated sentence at the same time it generates one word at a time. The decoder utilizes the final representation output by the encoder and every word it already output to generate the next word until it predicts and $< END >$ tag. In essence it will receive the sentence to be translated as input and will output the translated sentence one word at a time. Figs. 3.14 and 3.15 illustrate the architecture of the transformer model and the self attention mechanism.

The transformer architecture achieved state of the art performance in translation tasks and, due to its focus in attention, it was able to be trained significantly faster than other models centered around recurrent or convolutional layers. This led to a series of new models based on the transformer architecture and the idea that attention is all you need, one of its variants, a masked-language model composed of stacked Transformer encoders called BERT (Devlin *et al.*, 2018), quickly became the new baseline for NLP tasks such as (Rogers *et al.*, 2020), including classification tasks.

The BERT model is capable of considering the context of a word occurrence, differently than the previously mentioned context-free embeddings word2vec an GloVe. For example, the vector for "running" would have different embeddings in BERT for the phrases "He is running the company into the ground." and "He is running away" while word2vec would produce the exact same embedding.

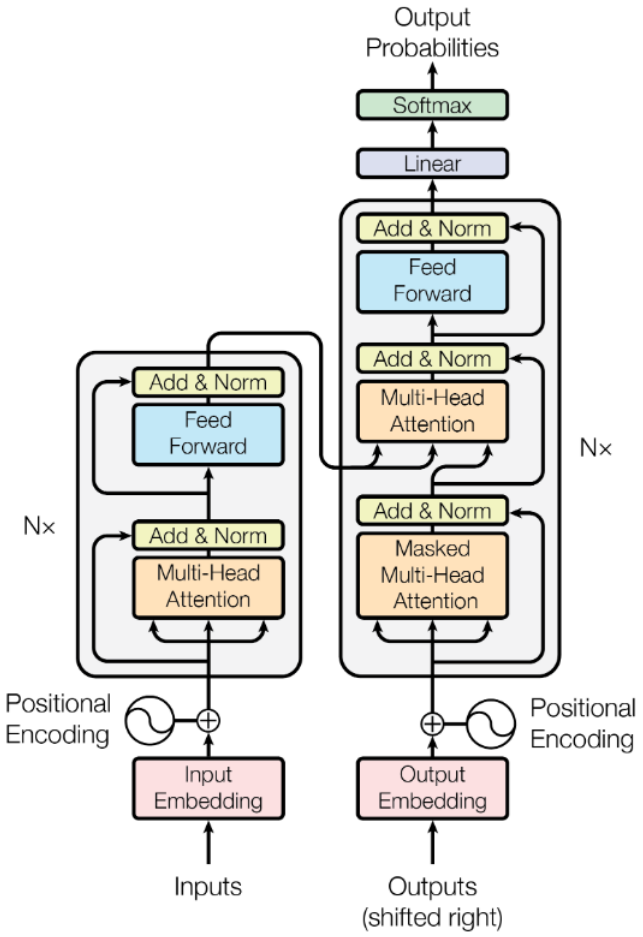


Figure 3.14: *The Transformer - model architecture. (Vaswani et al., 2017)*

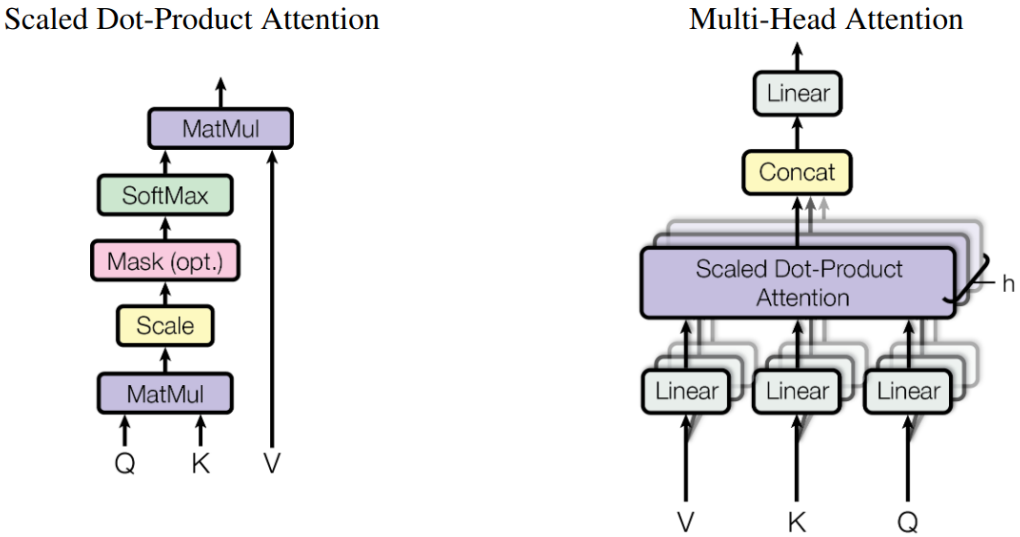


Figure 3.15: *(left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel. (Vaswani et al., 2017)*

3.7 Typical seq2seq Metrics

BLEU (Papineni *et al.*, 2002), or bilingual evaluation understudy, is a corpus wide NLP metric frequently used for sequence to sequence tasks such as machine translation or text summarization. This metric is a normalized score for *n*-grams overlap between the output and reference outputs (e.g. reference translations in a machine translation task) with a brevity penalty for shorter outputs.

Despite BLEU's wide spread use, it has some flaws. Among the main problems pointed out by the community are the metric's inability to take meaning and sentence structure into account, e.g. a rarer synonym could penalize a correct translation and a random shuffle of words of a correct translation could have a similar BLEU score albeit being complete nonsense (Callison-Burch *et al.*, 2006).

Another famous benchmark is the previously mentioned SQuAD 1.X (Rajpurkar, Zhang, *et al.*, 2016) and its successor SQuAD 2.0 (Rajpurkar, Jia, *et al.*, 2018), these are reading comprehension datasets that consist of questions posed by crowdworkers on a set of Wikipedia articles. The answers may or may not be contained in the corresponding reading passage upon which the question is theoretically posed, SQuAD 2.0 being composed of the questions from SQuAD 1.1 and 50,000 unanswerable ones written to look similar to answerable questions.

Chapter 4

Automated Refactoring

The "code processing community"[1] may be comparatively small to the NLP community, but it is not by any means non-existent. This chapter will present related work to our goal of automating function extractions, our objective is to situate the reader about recent relevant developments and to explicitate how this project differs from recent and current works in the area.

4.1 Rename Method

The rename method refactoring is one of the most commonly performed refactorings (Aniche *et al.*, 2020), an illustration of it can be seen in Fig. 4.1. Due to it essentially being an act of naming something for humans to read, it is a task that can be greatly benefited from natural language processing techniques and insights.

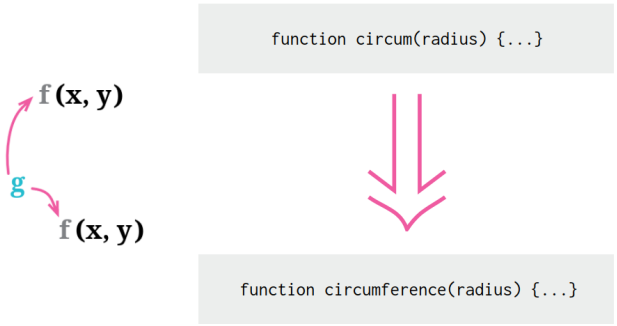


Figure 4.1: *Illustration of a refactoring of the Rename Method type. Illustration adapted from Martin Fowler (2023b).*

In this section we will explore a few recent works that attained state of the art performance in this task at the time of their publication. But since many of them utilize an AST to achieve that, we will start by explaining what an AST is.

[1] There does not seem to be a consensus for a name for the area, some common names are big code and programming language processing.

4.1.1 AST

For source code to become an executable program it needs to be translated into a low level language capable of being run by the machine or into machine code directly. A common approach is to write a compiler that will take the source code and compile it into an executable binary that may be used any time. A compiler is composed of numerous processing steps that can be somewhat separated into 2 categories: analysis and synthesis Aho *et al.* (2007). The analysis part pre-processes the inputted code and generates an intermediary representation of it to be used by the synthesis step. If during the analysis the code is found to be grammatically incorrect (in respect to the programming language formal grammar) or to have any other problem of lexical, syntactical or semantic nature it will abort the compilation process and report to the user an error message. By the end of a successful analysis step an intermediary representation of the source code will be generated, e. g. an Abstract Syntax Tree.

The AST is created by taking its concrete counterpart the concrete parse tree, also known as a parse tree, and removing any redundant or unnecessary information such as idiosyncrasies related to the source language or precedence and punctuation tokens (such as parenthesis in arithmetic expressions or ";" to denote the end of a statement) that are made unnecessary once the parse tree is built. Or as it is succinctly described in the book *Modern Compiler Implementation in Java*:

> *The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation.*
> Appel (2004)

It is no surprise that ASTs are often selected as a starting point for code embeddings, as it captures the structure of the source code that is not readily available when analyzing solely the source code. An AST can be seen in Fig. 4.2 and its corresponding source code in Listing 1.

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        int a, b;

        a = 3;
        b = 4;
        int c = a + b;

        // Random comments for illustrative purposes
        // private static int extracted(int a, int b) {
        // int c = a + b;
        // return c;
        // }
    }
}
```

Listing 1: *Small Java script, its AST can be seen in Fig. 4.2.*

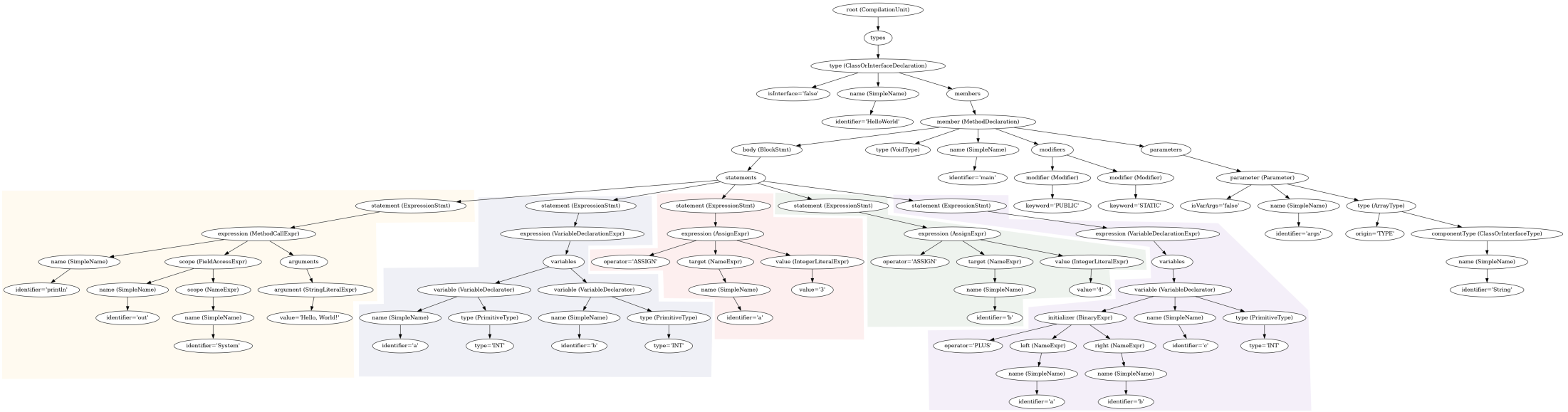


Figure 4.2: *AST of Listing 1. In an attempt to make the AST more intuitive and easier to read for those that never worked with them, we colored the sub-trees that correspond to the lines of the function body with one color per corresponding line*[a]*. AST printed through the* `dot` *utility and the* `JavaParser` *package, the code utilized to print the AST is available at Appendix A as Listing 2.*

[a] Since Java utilizes ";" to denote the end of a statement, multiple statements could be present in a single line of code but the AST would break down each line in its composing statements with one sub-tree for each. The existence of different lines is simply syntactic sugar, all Java programs could be expressed in single lines

4.1.2 code2vec and code2seq

Code2vec (Alon, Zilberstein, *et al.*, 2018) is an algorithm that generates code embeddings, i.e. distributed representation of code and similar in spirit to word2vec. A path-based attention model was developed for learning embeddings of arbitrary-sized snippets of code, which was done through the use of paths in the program's abstract syntax tree as a representation for code. The authors managed to leverage the underlying structure of the programming language to develop a scalable and efficient code embedding generator.

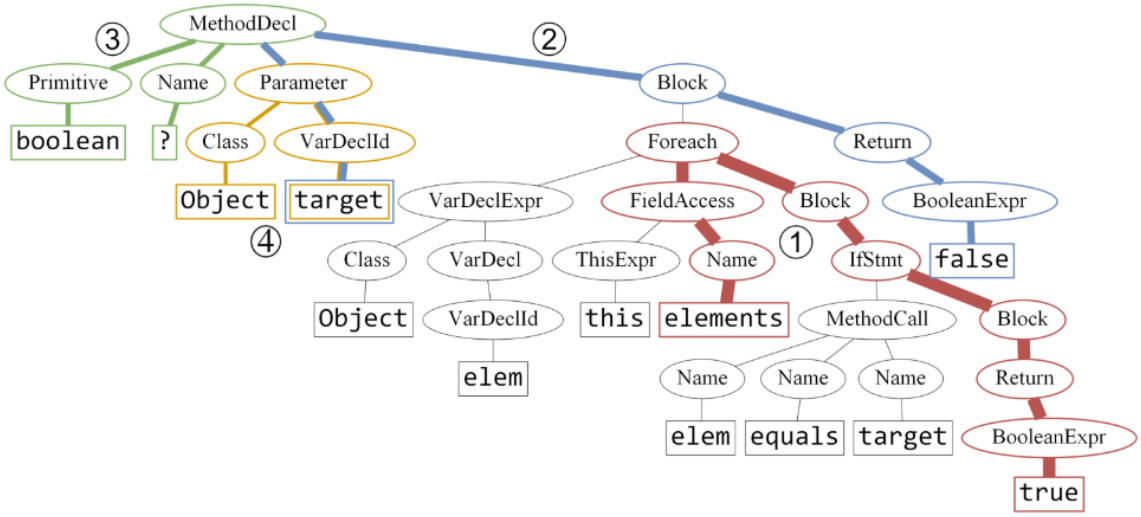


Figure 4.3: *Illustration of the path-based attention model from code2vec. The width of each colored path is proportional to the attention it was given (red 1: 0.23, blue 2: 0.14, green 3: 0.09, orange 4: 0.07). (Alon, Zilberstein, et al., 2018)*

Similarly to word2vec, this representation was successful in learning syntactic and semantic information, being able to generate analogies such as "*receive* is to *send* as *download* is to: *upload*".

Furthermore, code2vec was shown to be capable of capturing the resulting effect of two different function calls in succession, e.g. when given the embedding for the functions "equals" and "toLower", their sum is predicted to be equivalent to the function "equalsIgnoreCase", i.e. "$equals + toLower \approx equalsIgnoreCase$".

Code2seq (Alon, Brody, *et al.*, 2018) is another model created by the same group that built code2vec, building upon their previous findings and successes working with AST paths. It represents a code snippet as the set of compositional paths in its abstract syntax tree and uses attention to select the relevant paths while decoding. This new approach achieved better performance than code2vec, achieving a new state of the art while expanding the abilities of their model. It is capable of performing code summarization, caption and documentation.

4.1.3 Code Transformer

The code transformer model from Zügner *et al.* (2021) has a more holistic approach, instead of utilizing the pure source code or the AST it chooses to do both. By utilizing

what they define as context (source code) and structure (AST) they were able to combine two facets of programs and obtain state-of-the-art performance on monolingual code summarization in five languages and propose the first multilingual code summarization model. This multilingual model substantially outperformed its mono-lingual variants on all programming languages of the study.

The authors also noted that multilingual training only from context did not lead to the same improvements, highlighting the benefits of combining structure and context. An overview of the code transformer structure can be seen in Fig. 4.4.

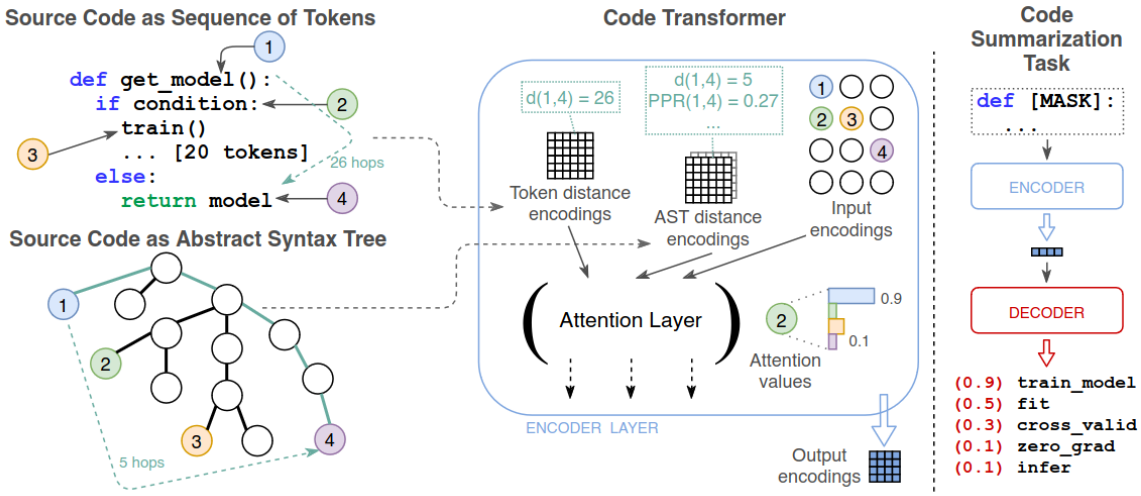


Figure 4.4: *"**Left:** Sequence (Context) and AST (Structure) representation of an input code snippet. **Center:** The CODE TRANSFORMER jointly leverages the sequence of tokens and the Abstract Syntax Tree to learn expressive representations of source code. In addition to the input token and node embeddings the model uses different distances between the tokens, e.g., shortest paths on the AST or personalized PageRank, to reason about their relative positions. The output embeddings can be used for downstream tasks such as code summarization **(right)**." (Zügner et al., 2021)*

4.2 Github Copilot X, Code Whisperer and Code Assistants

In recent years there has been an influx of code assisting tools based on machine learning models. Most of them are closed source with no details about their implementation, such as TabNine's autocompleter (TabNine, 2020), Amazon's CodeWhisperer code assistant (Amazon, 2023) and Microsoft's Copilot, or even platform specific such as Replit's Ghostwriter (Replit, 2023). With the release of Copilot X (GitHub, 2023), the latest iteration of the GitHub Copilot project, some details were made available to the public about its implementation but not that much is known besides the fact that they claim to use GPT-4.

Another interesting experimental project by the GitHub Next (2023) team are the code brushes , where the user can choose "brushes" to apply certain effects on code, such as making it more readable, adding types or including debugging statements, among

other code transformations. Most of its brush operations are essentially refactorings, small operations with the intent of making code more readable and maintainable without altering its behavior. However, currently there is not a brush for function extractions.

These "AI" assisted tools have managed to achieve great popularity among developers, however due to their closed source nature, wait lists and paywalls, it is difficult to even try to compare them with the work developed in this project since we lack access to them and the knowledge about what is going on under the hood. The more flexible systems based on LLMs, such as Copilot X, could be tailored through prompts to attempt to realize function extractions, but doing so in a programmatical manner to measure its performance would be a challenge. We are still awaiting for the opportunity to test the Copilot X but even if we had access to it and achieved a good performance in the function extraction task there would be the doubt about the nature of this performance: is the model performing well or is it simply overfitted? Since our training, validation and test sets come from Java projects in GitHub, it would be hard to determine if the test files were never used during the Copilot X training leading to data snooping.

For these reasons, there won't be any comparisons between these tools and the models we developed, however we felt the need to mention them for the sake of completeness.

4.3 Machine learning based code refactoring prediction

Work from Aniche *et al.* (2020) has shown that supervised machine learning methods are effective in predicting refactoring opportunities and, more importantly, such models can accurately model the refactoring recommendation problem. For this task, process and ownership metrics where shown to be essential for model creation, were the ownership metrics correspond to the suite proposed by Bird *et al.* (2011) and the process metrics are: quantity of commits, number of bug fixes, the sum of lines added, the sum of lines removed, and number of previous refactoring operations.

Albeit an important milestone, it's important to emphasize that the model proposed on this work was only capable of predicting refactoring *types* and *opportunities*, not the refactoring operations themselves. The models generated have also been shown to be robust with context change, i.e. a model trained in one context (e.g. Apache projects) can accurately predict refactoring opportunities on another context/project (e.g. F-droid projects). In Table 4.1 we present the performance of the different models trained by Aniche *et al.* (2020).

4.3.1 DataSet

Aniche *et al.* (2020) constructed a 40Gb dataset of source code from Java libraries, their entire Git commits history and metrics calculated on each commit. This was built upon 3 different ecosystems: Apache, F-droid and Github; an outline of the dataset can be referenced in Table 4.2. The Apache ecosystem is composed of all Java-based Apache

	Logistic Regression			SVM (linear)			Naive Bayes (gaussian)			Decision Tree			Random Forest			Neural Network		
	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc
Class-level refactorings																		
Extract Class	0.78	0.91	0.82	0.77	0.95	0.83	0.55	0.93	0.59	0.82	0.89	0.85	0.85	0.93	0.89	0.80	0.94	0.85
Extract Interface	0.83	0.93	0.87	0.82	0.94	0.87	0.58	0.94	0.63	0.90	0.88	0.89	0.93	0.92	0.92	0.88	0.90	0.89
Extract Subclass	0.85	0.94	0.89	0.84	0.95	0.88	0.59	0.95	0.64	0.88	0.92	0.90	0.92	0.94	0.93	0.84	0.97	0.89
Extract Superclass	0.84	0.94	0.88	0.83	0.95	0.88	0.60	0.96	0.66	0.89	0.92	0.90	0.91	0.93	0.92	0.86	0.94	0.89
Move And Rename Class	0.89	0.93	0.91	0.88	0.95	0.91	0.69	0.94	0.76	0.92	0.95	0.94	0.95	0.95	0.95	0.88	0.94	0.91
Move Class	0.92	0.96	0.94	0.90	0.97	0.93	0.67	0.96	0.74	0.98	0.96	0.97	0.98	0.97	0.98	0.92	0.97	0.94
Rename Class	0.87	0.94	0.90	0.86	0.96	0.90	0.63	0.96	0.69	0.94	0.91	0.93	0.95	0.94	0.94	0.88	0.94	0.91
Method-level refactorings																		
Extract And Move Method	0.72	0.86	0.77	0.71	0.89	0.76	0.63	0.94	0.69	0.85	0.75	0.81	0.90	0.81	0.86	0.79	0.85	0.81
Extract Method	0.80	0.87	0.82	0.77	0.88	0.80	0.65	0.95	0.70	0.81	0.86	0.82	0.80	0.92	0.84	0.84	0.84	0.84
Inline Method	0.72	0.88	0.77	0.71	0.89	0.77	0.61	0.94	0.67	0.94	0.87	0.90	0.97	0.97	0.97	0.77	0.85	0.80
Move Method	0.72	0.87	0.76	0.71	0.89	0.76	0.63	0.93	0.70	0.98	0.87	0.93	0.99	0.98	0.99	0.76	0.84	0.78
Pull Up Method	0.78	0.90	0.82	0.77	0.91	0.82	0.68	0.95	0.75	0.96	0.88	0.92	0.99	0.94	0.96	0.82	0.87	0.84
Push Down Method	0.75	0.89	0.80	0.75	0.90	0.80	0.66	0.94	0.73	0.97	0.76	0.87	0.97	0.83	0.90	0.81	0.92	0.85
Rename Method	0.77	0.89	0.80	0.76	0.90	0.80	0.65	0.95	0.71	0.78	0.84	0.80	0.79	0.85	0.81	0.81	0.82	0.81
Variable-level refactorings																		
Extract Variable	0.80	0.83	0.82	0.80	0.83	0.82	0.62	0.94	0.68	0.82	0.83	0.82	0.90	0.83	0.87	0.84	0.89	0.86
Inline Variable	0.76	0.86	0.79	0.75	0.87	0.79	0.60	0.94	0.66	0.91	0.85	0.88	0.94	0.96	0.95	0.81	0.82	0.82
Parameterize Variable	0.75	0.85	0.79	0.74	0.86	0.78	0.59	0.94	0.65	0.88	0.81	0.85	0.93	0.92	0.92	0.80	0.83	0.81
Rename Parameter	0.79	0.88	0.83	0.80	0.88	0.83	0.65	0.95	0.71	0.99	0.92	0.95	0.99	0.99	0.99	0.82	0.87	0.84
Rename Variable	0.77	0.85	0.80	0.76	0.86	0.79	0.58	0.92	0.63	0.99	0.93	0.96	1.00	0.99	0.99	0.81	0.84	0.82
Replace Variable With Attribute	0.79	0.88	0.82	0.78	0.89	0.82	0.64	0.95	0.71	0.90	0.84	0.88	0.94	0.92	0.93	0.79	0.92	0.84

Table 4.1: *The precision (Pr), recall (Re), and accuracy (Acc) of the different machine learning models, when trained and tested in the entire dataset (Apache + F-Droid + GitHub). Values range between [0,1] (Aniche et al., 2020). Table reconstructed from the original paper.*

software projects supported by the Apache Software Foundation (2023) and the F-droid (2023) ecosystem is a repository of FOSS Android mobile apps. Lastly, the GitHub ecosystem is composed of the first 10,000 most starred Java projects hosted on GitHub (removing duplicates of the Apache and F-Droid cohorts that are also hosted on GitHub as mirrors).

	Number of projects	Total number of commits
Apache	844	1,471,203
F-Droid	1,233	814,418
GitHub	9,072	6,517,597
Total	11,149	8,803,218

Table 4.2: *Number of projects and commits per ecosystem and in total.*

By utilizing RefactoringMiner (Tsantalis, Ketkar, *et al.*, 2020), the current state-of-the-art tool for refactoring identification in Java, on only the production files they were able to identify 20 different refactoring classes with varying number of occurrences, e.g. the dataset contains 327, 493 instances of the *Extract Method* refactoring class but only 654 of *Move and Rename Class*. Table 4.3 presents an overview of each refactoring instance obtained by different cohort and refactoring type.

At the same time that the refactoring instances were identified, non-refactoring instances and metrics for both of them were also obtained.

	All	Apache	GitHub	F-Droid
Class-level refactorings				
Extract Class	41,191	6,658	31,729	2,804
Extract Interface	10,495	2,363	7,775	357
Extract Subclass	6,436	1,302	4,929	205
Extract Superclass	26,814	5,228	20,027	1,559
Move And Rename Class	654	87	545	22
Move Class	49,815	16,413	32,259	1,143
Rename Class	3,991	557	3,287	147
Method-level refactorings				
Extract And Move Method	9,723	1,816	7,273	634
Extract Method	327,493	61,280	243,011	23,202
Inline Method	53,827	10,027	40,087	3,713
Move Method	163,078	26,592	124,411	12,075
Pull Up Method	155,076	32,646	116,953	5,477
Push Down Method	62,630	12,933	47,767	1,930
Rename Method	427,935	65,667	340,304	21,964
Variable-level refactorings				
Extract Variable	6,709	1,587	4,744	378
Inline Variable	30,894	5,616	23,126	2,152
Parameterize Variable	22,537	4,640	16,542	1,355
Rename Parameter	33,6751	61,246	261,186	14,319
Rename Variable	324,955	57,086	250,076	17,793
Replace Variable w/ Attr.	25,894	3,674	18,224	3,996
Non-refactoring instances				
Class-level	10,692	1,189	8,043	1,460
Method-level	293,467	38,708	236,060	18,699
Variable-level	702,494	136,010	47,811	518,673

Table 4.3: *The number of instances of refactoring and non-refactoring classes used in Aniche et al. (2020). Table reconstructed from the original paper, our emphasis.*

Chapter 5

(Re)Building a Dataset

Originally, this project was conceived as a partnership with the TUDelft where we would build upon their previous work by training our models on the dataset they build in Aniche *et al.* (2020) and that we previously presented in Section 4.3.1. However once we received the full dataset we realized that it was not suitable for our needs. Since they did not capture any information regarding which lines were extracted in the function extractions nor any other more minute details about where and how exactly the refactorings occurred, there was no way to train a supervised machine learning model. The dataset could still be used for unsupervised approaches and other less granular tasks, however we decided to collect our own dataset to be able to accomplish our original objectives. We briefly explored the possibility of leveraging the existing dataset to obtain the extracted lines but we reached the conclusion that scrapping them from scratch would be simpler and faster. This heavily impacted our proposed timeline since there was a need to create an usable dataset from scratch. Utilizing a list of over 40.000 Java repositories provided by SERG - TUDelft, and made available at Refactoring.ai (2021), we created a data pipeline with a tool called RefactoringMiner (Tsantalis, Ketkar, *et al.*, 2020) at its core. In this chapter we will detail how we constructed this dataset and the reasoning behind some of our decisions, this new objective of creating a dataset instead of using a pre-built one led us to a series of new decisions. The first of them being our decision to maintain our objective of working with function extractions. If we reference Table 4.3 one might notice that the most common refactoring was of the *Rename Method* type. Although they were not originally published with this sole task or problem formulation in mind, code2vec, code2seq and code transformer achieve an impressive performance in this task (Alon, Zilberstein, *et al.*, 2018; Alon, Brody, *et al.*, 2018; Zügner *et al.*, 2021). In contrast, the second most common refactoring was of the type *Extract Method* and to the best of our knowledge there is no model for automating this refactoring. Tackling this new and interesting task was the biggest motivator behind our choice of working with function extractions instead of other less common refactorings.

This leads us to the next decision in our data building task, what tool should we use to obtain function extractions?

5.1 RefactoringMiner

To create a model for automatic function extraction our dataset needs to have examples of function extractions with the lines extracted being appropriately tagged. Doing so by hand would be error prone and time consuming, so tools crafted to "mine" git repositories for such refactorings are commonly used.

One of such tools is RefactoringMiner 2.0. It can detect 40 different refactoring types through rules based on statement mapping information and AST node replacements. In its 1.0 version (Tsantalis, Mansouri, *et al.*, 2018) it attained a state of the art performance and, maybe more importantly, introduced a refactoring oracle of validated refactoring instances, providing a new rigorous benchmark for the field. RefactoringMiner 2.0 expanded this oracle into 7,226 true positives in total, for 40 different refactoring types detected by one (minimum) up to six (maximum) different tools. These refactorings were mined from 536 commits from 185 open-source Java projects hosted on GitHub. Another important feature present in RefactoringMiner 2.0 is the ability to detect nested refactorings:

> *A refactoring operation that takes place in code resulting from the application of another refactoring operation is a nested refactoring. For example, the renaming or extraction of local variables inside the body of an extracted method are nested refactoring operations[...].*
> Tsantalis, Ketkar, *et al.* (2020)

Fig. 5.1 shows a real case of nested extract method refactorings.

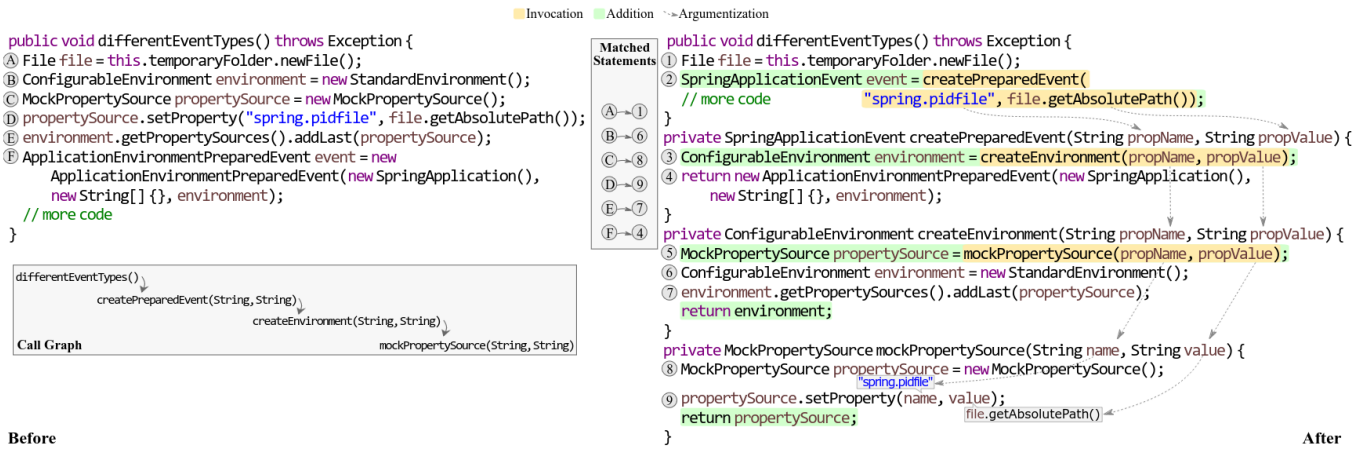


Figure 5.1: *Nested extract method refactorings mined from github.com/spring-projects/spring-boot/commit/becce. There are 3 levels of nested extracted methods with each extracted method calling the subsequent one. Image from Tsantalis, Ketkar, et al. (2020).*

This tool is the same one utilized to create the dataset described in Section 4.3.1, so when we were choosing a refactoring detection tool it was naturally considered. But with the release of new tools and versions we were interested if RefactoringMiner could still be considered the state of the art in all Java refactoring detections so we explored the possibility of using other tools such as RefDiff (Silva *et al.*, 2020) and RefDetect (Moghadam *et al.*, 2021). However, we found that specifically for function extractions RefactoringMiner

still outperforms its competitors, as can be seen in Tables 5.2 and 5.1.

		RefactoringMiner 2.0		RefactoringMiner 1.0		RefDiff 0.1.1		RefDiff 1.0		RefDiff 2.0	
Refactoring Type	**#TP**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
Inline Method	107	100	95.3	97.8	85.0	81.0	79.4	97.1	61.7	88.8	66.4
Extract Method	957	99.8	95.8	97.6	75.5	94.6	79.3	98.7	39.8	98.5	61.4
Move Field	249	98.4	96.0	84.1	93.6	32.3	42.6	55.5	44.6	N/A	
Move Class	1091	100	99.3	99.5	93.4	99.3	89.9	99.6	95.8	99.8	97.3
Extract Interface	21	100	100	95.0	90.5	80.0	57.1	61.1	52.4	87.5	100
Push Down Method	43	100	97.7	100	74.4	100	44.2	100	44.2	87.8	83.7
Push Down Field	33	100	100	100	78.8	100	90.9	100	78.8	N/A	
Pull Up Method	291	100	97.9	100	93.5	97.6	28.2	97.1	22.7	96.4	92.1
Pull Up Field	129	100	99.2	100	98.4	100	24.0	100	17.8	N/A	
Move Method	283	99.6	92.9	96.2	79.9	28.6	91.5	58.7	90.8	76.2	80.2
Rename Method	371	98.2	87.9	94.7	67.4	85.8	76.5	94.4	59.6	90.9	62.3
Extract Superclass	71	100	100	100	100	100	18.3	89.3	70.4	98.1	71.8
Rename Class	56	100	89.3	97.4	67.9	92.3	85.7	97.7	76.8	96.1	87.5
Extract & Move Method	166	100	54.2	73.3	19.9	65.6	75.9	91.5	25.9	63.7	39.2
Move & Rename Class	41	100	90.2	70.0	51.2	74.2	56.1	92.0	56.1	80.0	68.3
Move & Inline Method	19	100	73.7	N/A		92.9	68.4	100	15.8	100	57.9
Average	3938	99.7	94.2	96.5	81.3	72.9	73.1	88.3	60.7	93.8	76.9

Table 5.1: *Precision and recall per refactoring type. Values calculated based on a refactoring oracle of validated instances containing 7,226 true positives in total, for 40 different refactoring types detected by one (minimum) up to six (maximum) different tools. Table and caption from Tsantalis, Ketkar, et al. (2020), our highlight.*

		RefDetect 1.0			RMiner 2.0.1		
Refactoring Type	**#TP**	**Precision**	**Recall**	**F-Score**	**Precision**	**Recall**	**F-Score**
Rename Method	356	95.7	74.9	84	97.4	76.4	85.6
Move Method	198	91.8	91.8	91.8	98.8	90	94.2
Push Down Method	36	100	66.7	80	100	91.2	95.4
Pull Up Method	285	99.6	90	94.6	99.6	96.8	98.2
Extract Method	707	98.4	86.7	92.2	98.8	90.8	94.6
Inline Method	180	98.2	91.1	94.5	98	54.4	70
Change Method Parameters	1272	99.2	94.8	97	99	88.9	93.7
Move & Rename Method	48	84.3	91.5	87.8	100	54.3	70.4
Extract & Move Method	190	92.4	57.4	70.8	98.6	39	55.9
Move & Inline Method	32	82.1	71.9	76.7	100	40	57.1
Move & Change Method Parameters	33	93.9	96.9	95.4	100	71.9	83.7
Method-Level Refactorings	**3337**	**94.1**	**83**	**87.7**	**99.1**	**72.2**	**81.7**

Table 5.2: *Precision, recall and f-score results per method-level refactoring type. Values calculated based on a refactoring oracle of validated instances from Tsantalis, Ketkar, et al. (2020), containing 7,226 true positives in total for 40 different refactoring types detected by one (minimum) up to six (maximum) different tools. Table from Moghadam et al. (2021), our highlight.*

With this, we also decided to utilize RefactoringMiner to build our dataset, but if in future work we expand on the refactorings we deal with or the targeted programming languages, RefDetect, a multilingual refactoring detection tool, may be a better pick:

> *RefDetect clearly outperformed R**(efactoring)**Miner in method and class based refactorings, achieving f-scores respectively of 87.7% vs. 81.7% for method-level refactorings and 92.1% vs. 86.9% for class-level refactorings.*
> Moghadam *et al.* (2021)

5.2 Pipeline

The overall structure of our pipeline is quite simple, as can be seen in Fig. 5.2. We start by cloning over 40.000 repositories of Java projects, these projects come from a list curated by the SERG group of the TUDelft university and available at Refactoring.ai (2021). With the repositories cloned we can start mining refactorings with RefactoringMiner. RefactoringMiner outputs one json file per mined repository, so our next step is to process all these json files and filter only the relevant features into our SQLite database. For this project we are only interested in function extraction refactorings, so we drop every other refactoring found. Then we check if the function extraction found covers a continuous span of lines, and if they are not continuous if maybe only blank lines or comments separate the continuous chunks; Fig. 5.3 illustrates this process with a Discrete Finite Automata. Lastly, we exclude any refactorings that happen at the same function in a same git commit. We do so to simplify our model and training process, we believe keeping these refactorings would negatively impact our performance and raise an issue of non determinism since multiple predictions could be found to be correct.

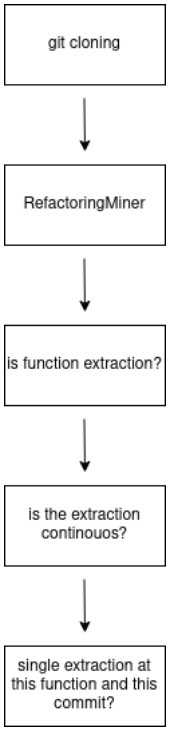


Figure 5.2: *A short visualization of the steps present in our data pipeline.*

The entirety of the code utilized in the construction of this dataset is available in Appendix B.

5.3 Exploration of the dataset

Even though we started with a list of 49,982 Java repositories, we only managed to obtain function extractions from 19,936 of them. Many factors played a part in this, such as

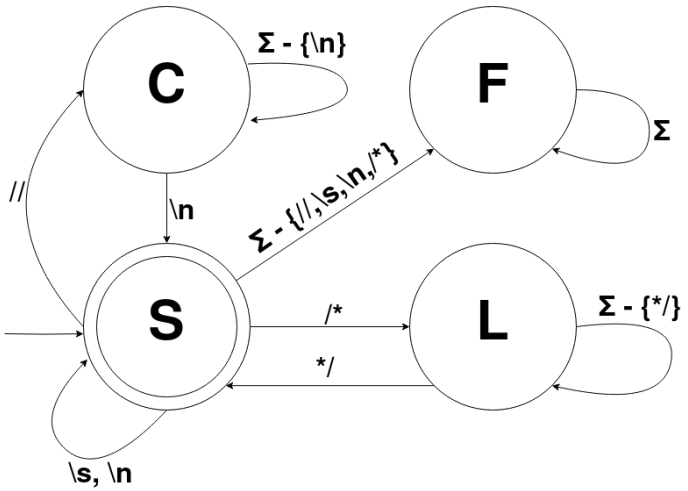


Figure 5.3: *A Discrete Finite Automata that detects if any of the lines analyzed is not a comment or blank line. For the sake of simplicity and illustration, let us consider that symbols such as "//", "/*", "*/", "\n" and "\s" are single characters. Building a real DFA that breaks each of these "signals" into their constituent characters would increase the complexity of the system and loose its meaning as an illustration to clarify our data processing. Following convention, Σ represents the alphabet of this DFA, i.e. the set of all valid characters in the java language. State **S** represents blank lines, the **C** state represents comments and **L** long comments, lastly the **F** state represents a failure, once something that does not constitute a comment, long comment or blank line is detected the process gets stuck in the **F** state unable to ever reach the accepting state **S**.*

problematic encodings when processing the files, repositories not found due to renaming or migration, RefactoringMiner limitations, only refactorings other than function extraction found or even repositories that weren't purely written in Java (some were mostly written in Kotlin with just a few files in Java, for example).

However, even with all these issues we obtained 523,667 different instances of function extraction, an increase of over 60% in comparison with the dataset we originally intended to use from Aniche *et al.* (2020). Interestingly, over 80% of the function extractions found are continuous, given the diverse range of Java projects used in this analysis we believe this should be a good approximation to the real proportion of continuous function extractions in relation to non-continuous. This further solidified our decision to focus on only continuous extractions for our model, since they are not only simpler to train but also arguably a more useful and actionable suggestion for developers.

Chapter 6

Models

In this chapter we will be exploring the different components of the models we developed and the motivation behind our decisions. Lastly the hyper-parameters choice and the optimization of the different models trained will also be briefly explored.

6.1 Embeddings

To create our function extraction models we need to be able to calculate an embedding for code functions. However, none of the previously explored code embeddings in Chapter 4 are granular enough for our needs, they expect whole functions while we need to be able to at the very least create embeddings per line of code. Exploring the literature, in particular Allamanis, Earl T Barr, *et al.* (2018) which presents a table of over 30 code embedding generators for a diverse range of tasks, we were not able to find a fitting embedding. One of our original objectives was to develop our own code embedding for this task, however due to time constraints caused by the dataset delay this was no longer feasible.

So, once again leveraging the idea of naturalness of programming languages that we introduced in Chapter 1 we opted to utilize embeddings for natural languages — in particular english — to represent our code.

There were two main types of embeddings tested in this project: the more traditional GloVe based embeddings and the transformer based embeddings that we had the opportunity of exploring during the semester abroad at TUDelft thanks to the FAPESP BEPE program. Both embeddings are sentence embeddings, i.e. each line of code will have its own vector embedding. Here we will give a short overview of how these embeddings work and how they were built.

Transformer based

Following our initial goal of exploring the use of transformers for refactoring tasks we arrived at two main approaches: utilizing transformers in our embedding and/or as our model. However, due to our time constraints imposed by our need to create our dataset from scratch, we focused on utilizing transformer based embeddings as the first step since

adapting the dataset to be compatible with training a transformer model would be an extremely manual and time consuming process. Another relevant point is the maxim of "garbage in garbage out", that essentially states that if your data is bad your results will also be bad. We concluded that it would be more beneficial to first make sure that we have at least one reasonable embedding for our code before we start investing such a huge amount of our scarce time in adopting this new architecture. Unfortunately, by the time we finished exploring the use of transformer based embeddings, there was no feasible way of training a transformer model in the remaining time of the BEPE project. Therefore, in this section we will be exploring the use of transformer based embeddings and their performance compared to more traditional embeddings.

Among the many embedding options available for our analysis we chose to utilize the transformer based models of the SBERT project (Reimers and Gurevych, 2019). Until recently this project was referred to as the state of the art in many tasks involving sentence embeddings, furthermore it possesses embeddings trained through a diverse number of transformer based architectures with most of them being more computationally efficient when compared to other models such as BERT (Devlin *et al.*, 2018) or RoBERTa (Liu *et al.*, 2019).

In Fig. 6.1, also available as an interactive table at Nils Reimers, Iryna Gurevych (2022b), we can see the main 13 models of interest available in the SBERT project and some metrics regarding them. Due to our time constraints we were forced to select only a few of these embeddings for our experiment, since they may take an entire day to train a single epoch. From this list we discarded all models solely trained on Q&A datasets since they are too distinct from our real objective and other models that were too derivative of another model already present in the list such as *all-MiniLM-L6-v2* (half the layers of *all-MiniLM-L12-v2*), *distiluse-base-multilingual-cased-v2* (trained in an additional 35 languages in comparison to *distiluse-base-multilingual-cased-v1* but since most of our Java files are believed to be written in english we don't believe this would bring us significant gains in performance) and *paraphrase-multilingual-mpnet-base-v2* (a slow and more inefficient version of *all-mpnet-base-v2* that was trained in a smaller and more specific dataset).

After short-listing the available embeddings we were left with 7 embeddings to test out:

- all-mpnet-base-v2 (Song *et al.*, 2020) (Model Card)
- all-distilroberta-v1 (Liu *et al.*, 2019; Sanh *et al.*, 2019) (Model Card)
- all-MiniLM-L12-v2 (W. Wang *et al.*, 2020) (Model Card)
- paraphrase-albert-small-v2 (Lan *et al.*, 2019) (Model Card)
- paraphrase-multilingual-MiniLM-L12-v2 (W. Wang *et al.*, 2020) (Model Card)
- paraphrase-MiniLM-L3-v2 (W. Wang *et al.*, 2020) (Model Card)
- distiluse-base-multilingual-cased-v1 (Sanh *et al.*, 2019) (Model Card)

Model Name	Performance Sentence Embeddings (14 Datasets)	Performance Semantic Search (6 Datasets)	Avg. Performance	Speed	Model Size
all-mpnet-base-v2	69.57	57.02	63.30	2800	420 MB
multi-qa-mpnet-base-dot-v1	66.76	57.60	62.18	2800	420 MB
all-distilroberta-v1	68.73	50.94	59.84	4000	290 MB
all-MiniLM-L12-v2	68.70	50.82	59.76	7500	120 MB
multi-qa-distilbert-cos-v1	65.98	52.83	59.41	4000	250 MB
all-MiniLM-L6-v2	68.06	49.54	58.80	14200	80 MB
multi-qa-MiniLM-L6-cos-v1	64.33	51.83	58.08	14200	80 MB
paraphrase-multilingual-mpnet-base-v2	65.83	41.68	53.75	2500	970 MB
paraphrase-albert-small-v2	64.46	40.04	52.25	5000	43 MB
paraphrase-multilingual-MiniLM-L12-v2	64.25	39.19	51.72	7500	420 MB
paraphrase-MiniLM-L3-v2	62.29	39.19	50.74	19000	61 MB
distiluse-base-multilingual-cased-v1	61.30	29.87	45.59	4000	480 MB
distiluse-base-multilingual-cased-v2	60.18	27.35	43.77	4000	480 MB

Figure 6.1: *Table of 13 embedding models available in the SBERT project and some metrics regarding them. (Nils Reimers, Iryna Gurevych, 2022b)*

GloVe

GloVe (Pennington *et al.*, 2014) is a more traditional model that combines the advantages of two model families, local context windows and global matrix factorization. The model produces a vector space with meaningful sub-structure by leveraging statistical information in an efficient manner when training only on the nonzero elements in a word-word co-occurrence matrix, rather than on individual context windows in a large corpus or on the entire sparse matrix.

At the time this method was published it attained state-of-the-art performance on the word analogy task, and outperformed other methods (such as word2vec) on several word similarity tasks.

By utilizing GloVe we also have an interesting counter-point to our transformer based embedding, we are essentially comparing an older and well established model that is still relevant even in this ever evolving field (the same cannot be said about bag-of-words for example) to what is essentially the more recent and generally successful approach.

By taking the average of the word embeddings of a sentence it is possible to generate an embedding for that sentence. This is what was done by Nils Reimers, Iryna Gurevych (2022a) in order to generate two GloVe sentence embedding models, one trained with 6 billion parameters and the other with 840 billion parameters.

6.2 Architecture

An important issue to be addressed here comes from a fundamental difference between natural languages and programming languages: even if a sentence in english has a gram-

matical mistake it is capable of transmitting meaning, of carrying a semantic value. The same cannot be said about a grammatically incorrect piece of code. For this reason we are not able to blindly apply seq2seq NLP models on source code, we need to ensure that the code transformations we generate will not grammatically break the inputted code. To ensure the grammatical correctness of our refactoring we briefly explored the possibility of using language servers to intermediate our operations in Section 2.2.3.

Given this, our models simply needs to predict the line number of the start and end of the extracted linespan.

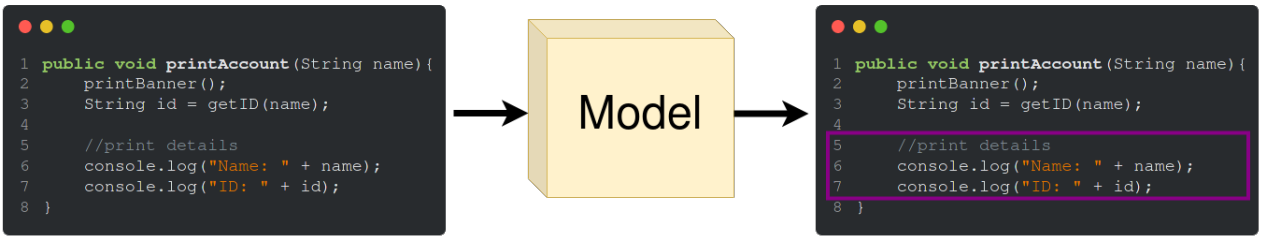


Figure 6.2: *Our model will receive the source code of a function definition that needs to be refactored and will output the line span that needs to be extracted. In this particular example the lines 5 to 7 of the `printAccount()` function need to be extracted.*

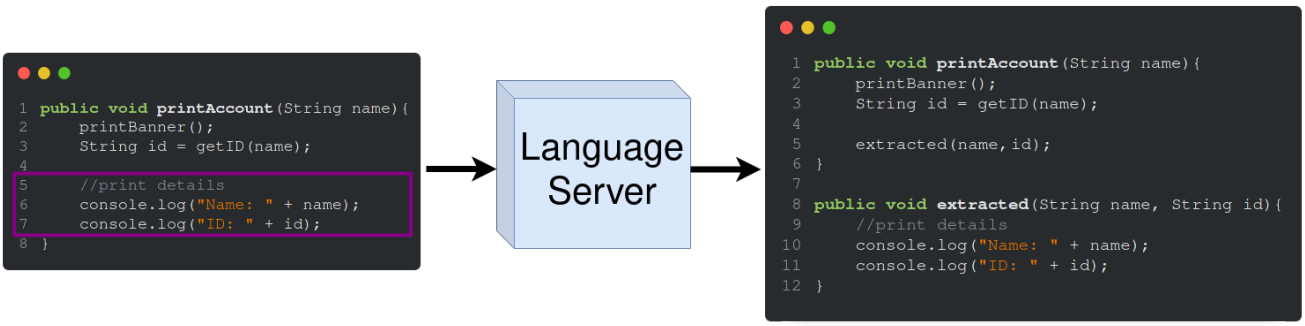


Figure 6.3: *Once the line span of the extraction has been determined, the language server is contacted and it will be responsible for realizing the extraction and returning the refactored code. In this example the lines 5 to 7 are refactored by the language server and become the function `extracted()`.*

In short, the model will be fed with the lines of a function one by one and it will generate as an output 2 pointers, one indicating the first line to be extracted and a second pointer indicating the last line to be extracted. Together these 2 pointers will define the line span to be refactored. Fig 6.2 presents a general illustration of our models, working as a black box, and Fig 6.3 illustrates the refactoring done through a language server with the output of the black box model from Fig. 6.2.

Simple RNN

Our first architecture is extremely simple; it consists of a LSTM layer that receives the embedded inputs and then passes on its hidden state to two different feedforward linear layers which will predict the start and end lines of the function extraction. As for the loss we chose the L1 distance function.

Pointer Network (Ptr-Net)

Pointer Networks (Vinyals *et al.*, 2015) are a flexible encoder decoder model that utilizes attention to select one of the inputs as an output at each decoding step. As mentioned in Section 3.5, this model has been used in a variety of tasks that involve re-ordering and/or selecting elements of the inputted data. This is particularly useful in our case since our objective can be formulated as selecting which lines of an inputted function that need to be extracted, in particular at which line the extraction begins and ends.

Our implementation of the pointer network utilizes a 1 layer LSTM as the encoder and another 1 layer LSTM as the decoder with the additive attention mechanism from Bahdanau *et al.* (2014a) tying them together. Lastly, to convert from the attention scores into the predicted lines we apply the softargmax function in order to apply the argmax function in a differentiable manner.

$$softargmax(x) = \sum_i \frac{e^{\beta x_i}}{\sum_j e^{\beta x_j}} i$$

β essentially controls how well this function approximate the argmax function, however a high enough β will approximate it so well that back-propagation will start to fail. In Appendix C we present a brief exploration of the impact of β on the model capability to learn. Luckily, our experiments did not detect any significant increase in performance by increasing β over the value of 10, which is well below the point were back-propagation starts to fail.

We opted to use this function instead of the more commonly utilized softmax function in order to use the L1 loss instead of negative log likelihood loss. Since we are not predicting simple classes as our output, the error from missing the start of a refactoring by one line should be smaller than that of missing by 10 lines and this isn't possible with negative log likelihood. In Fig. 6.4 we can see an illustration of how this model works, particularly how the attention mechanism "chooses" the start and end of a function extraction.

6.3 Metrics

When dealing with a well researched topic one is able to build upon previous knowledge and conventions in the area. However, to the best of our knowledge, there is no previous research that aims to automate the task of function extractions so we are faced with the challenge of defining how to best evaluate the quality of a predicted refactoring. In Section 3.7 we explored a few common seq2seq metrics of NLP and their limitations, however none of the metrics presented are applicable for this particular use case, neither are the other metrics we could find in the literature. They may not be applicable to our particular use case but we may still take insights from them, such as BLEU's limitations that do not take into account if a sentence actually makes any semantic or grammatical sense.

So without being able to resort to a readily available NLP metric we turned ourselves to typical code quality metrics, such as the ones presented in Section 2.2.1. Albeit these

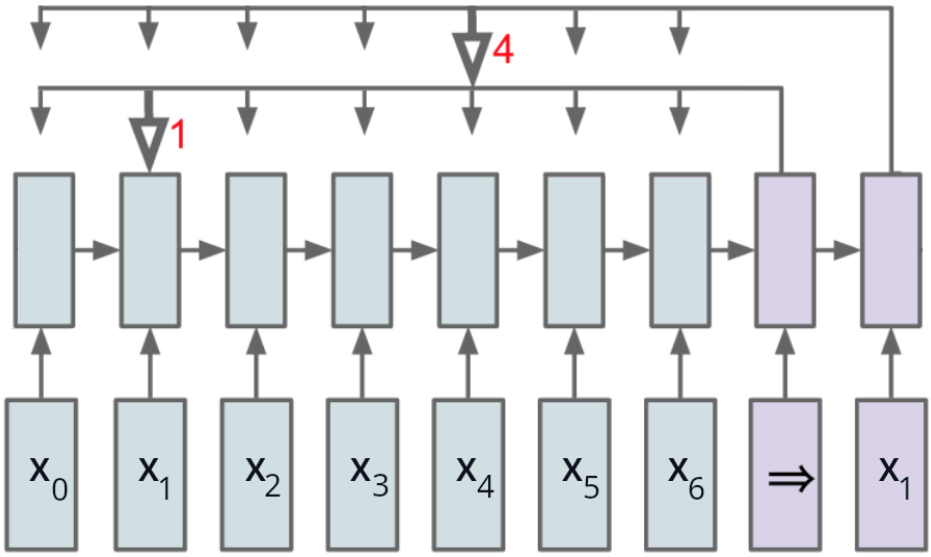


Figure 6.4: *An illustration of how the model works, with green blocks representing the encoder and the purple ones the decoder. Each x_i value fed to the encoder represents the embedding of a single line from the function being analyzed. The decoder receives as an input the hidden state from the last step (if available) and all the encoder hidden states, so to predict the last line to be extracted the decoder will receive all the hidden states from the encoder and the hidden state from last step that represents the first line to be extracted. Lastly, the attention mechanism is represented by the arrows pointing into the different encoder hidden states/inputs, the arrows may be seen as the final output of the model after the attention scores go through the softargmax function. So in this particular example being illustrated, the function is composed of 7 lines and it should go through an extraction of lines 1 through 4.*

metrics can be useful in certain occasions we found them to be lacking for our needs. They can present a score of the code quality but by their very nature they are subjective since they try to give a concrete score to an abstract and subjective concept like code quality. Simply deciding which metric to use can bring vastly different results since not all of them are tailored towards the same issues and they do not operate on the same scales, comparing the pure score of different code metrics is akin to comparing apples to oranges. Apart from that, our model performs only function extractions, the only code operation would be a transfer of code lines from an existing function to a newly created one, this would not bring a big impact in most code quality metrics. We could directly measure the size of functions instead of relying in more complicated code metrics but this would pose a new problem on itself, fewer lines of code does not necessarily equate to a better piece of code nor does a bigger extraction imply a better refactoring than a shorter one. Besides, these metrics are not very actionable, they do not always present a clear path on how to improve the refactoring to maximize them.

Another early idea was to leverage unit tests to verify code integrity, however it is not guaranteed that all functions being refactored have unit tests nor that we would be able to easily identify them in a programmatic manner. Also, unless the linespan breaks a loop or some other flow control structure (which could be easily detected) the extraction would necessarily be a code preserving transformation so code integrity is not a concern.

Lastly we turned to classic ML metrics, such as accuracy. We decided to utilize the

binary accuracy, which is essentially equivalent to jaccard score, to measure how close the overlap between the predicted linespan and the ground truth is.

The jaccard coefficient is a measure of similarity between sets, defined as (Jaccard, 1912):

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

An illustration of the jaccard index can be seen in Fig. 6.5.

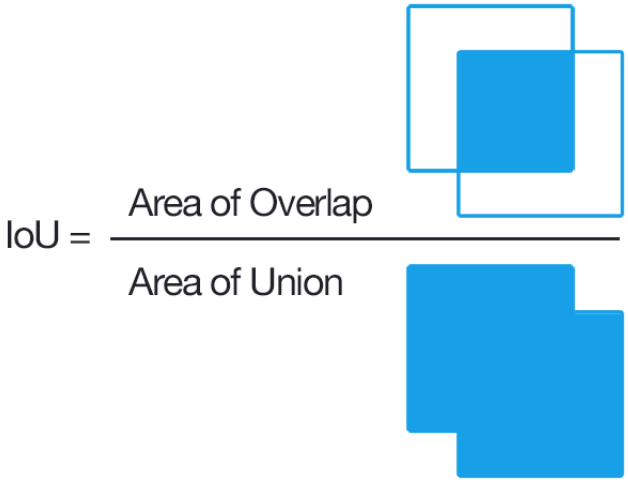


Figure 6.5: *A visual representation of the jaccard index. Image from Adrian Rosebrock (2016).*

Lastly, if an IDE plugin were to be implemented there would be the possibility of utilizing telemetry to evaluate user response to suggested refactorings. This would be a way to grasp the quality of suggested refactorings and to better understand when and why our model fails, however this is left for future work since such an endeavor was unfeasible given the time constraints of this project.

6.4 Hyper-parameter choice

Many scientific fields currently suffer from a reproducibility crisis (Baker, 2016) and machine learning is no exception (Heil *et al.*, 2021; Hutson, 2018; Gibney, 2022; Kapoor and Narayanan, 2022). So in the interest of transparency and reproducibility we include this section to explicitate how we arrived at our models hyper-parameters, i.e. the motivation and constraints behind our choices and the techniques used.

Originally we intended to optimize every single one of our models to the best of our ability in order to obtain the best performance possible, given our choices of architecture and embeddings. However due to time constraints this was simply not feasible, for instance training one epoch of some of the models that used transformer-based embeddings could take more than a whole day of training. That's why we had to compromise and decide exactly what was feasible to optimize and what would bring the greatest chance of improving the performance of our models.

We ended up deciding to optimize our pointer network models that utilized the GloVe based embeddings and when applicable to extrapolate our findings into the other models. More specifically we decided to optimize the following hyperparameters:

- Learning rate
- Weight decay
- Embedding
- Batch size
- Hidden size

To perform this optimization we split our dataset into training, validation and test and then we utilized the Optuna library from Optuna team (2022) to optimize our model by training many trial runs on the training set and testing their performance in the validation set.

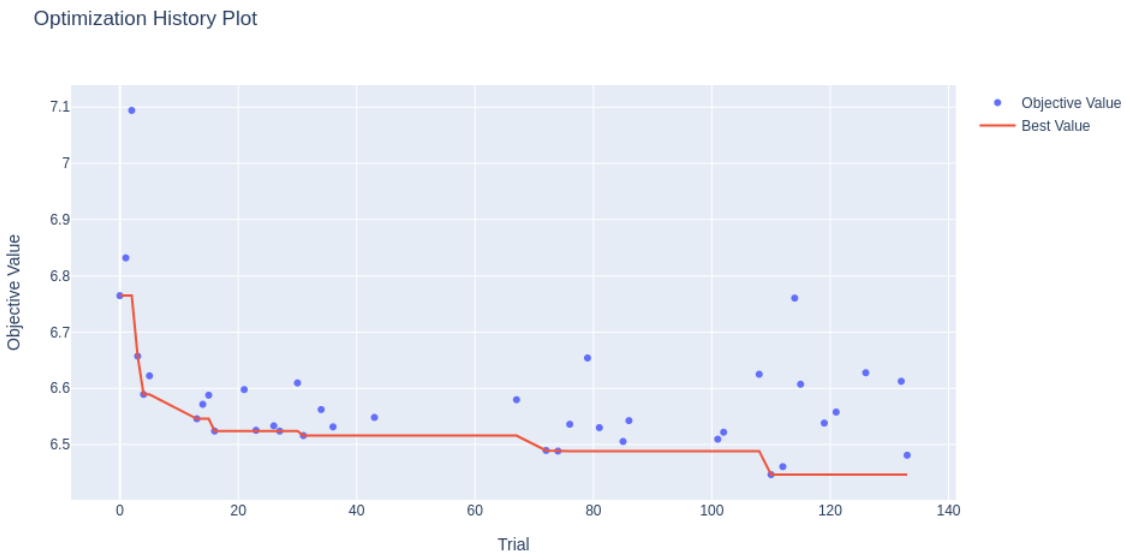


Figure 6.6: *Optimization history plot of the 136 Optuna trials.*

After a warm-up period this library is capable of pruning any given run that is not achieving a satisfactory performance, by doing this it is capable of exploring the parameter space in a more time efficient manner by only completing trial runs that have the possibility of achieving a better performance.

There are many different samplers available in this toolbox, which essentially help us explore the parameter space by sampling different parameter values, and between all of them we chose to utilize the TPE (Tree-structured Parzen Estimator) (Bergstra *et al.*, 2011). An important point of note about this sampler is that it assumes that the different hyperparameters are independent so it determines the value of a single parameter without considering any relationship between parameters. If this assumption is false the optimization process may take more time or in some cases even miss some opportunities

for further improvement when such hyperparameters have a strong relationship. We hypothesize that the hyperparameters that we chose are independent or at the least have a weak impact on one another but since we have never tested this hypothesis it is possible that our model could be further optimized by simply picking a more robust sampler. Due to our time constraints we were unable to further explore this point.

Fig. 6.6 presents the plot with the optimization history of our study and it's 196 trial runs and Fig. 6.7 presents the loss and accuracy plots of these trials.

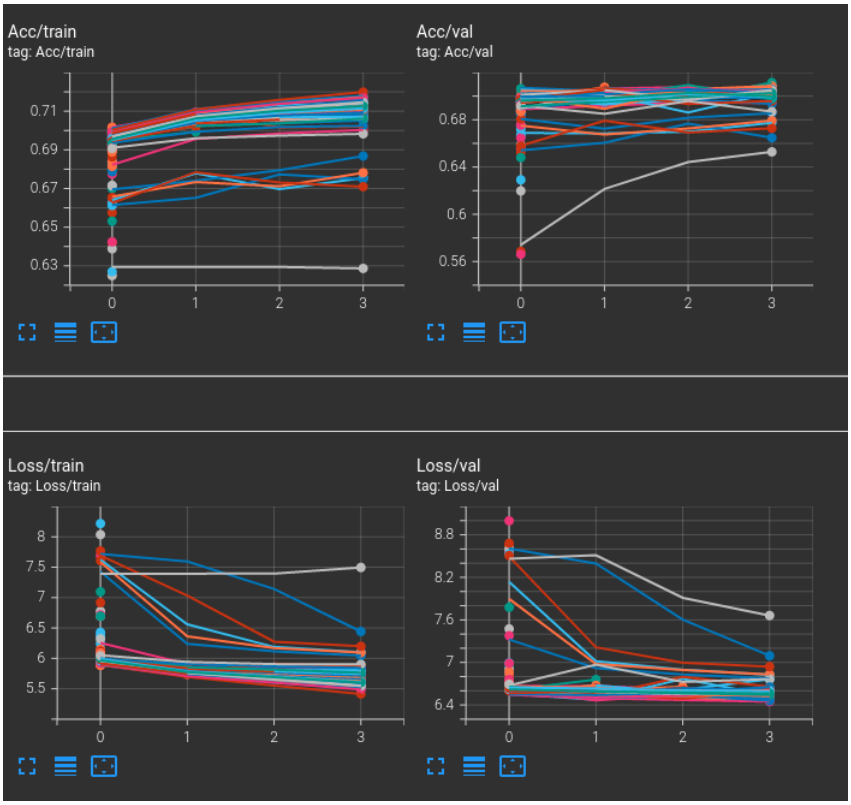


Figure 6.7: *Loss and accuracy plots of the 196 Optuna trial runs.*

In Fig. 6.8 and Fig. 6.9 we are able to see the slice plots of our parameters. Originally we were going to explore all parameters at the same time in a single study, but while we were still learning how to use Optuna it became clear that the GloVe embedding trained with 840 billion parameters could achieve a better performance over it's counterpart trained with only 6 billion parameters so we decided to exclude the embedding choice from the search space.

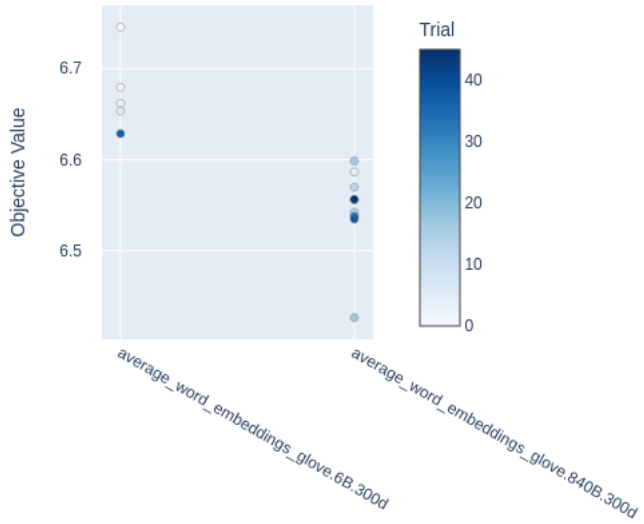


Figure 6.8: *Slice plot of 43 Optuna trial runs, since the GloVe embedding trained with 840 billion parameters could achieve a better performance over it's counterpart trained with only 6 billion parameters we decided to exclude the embedding choice from the search space from our subsequent runs, as can be seen in Fig. 6.9.*

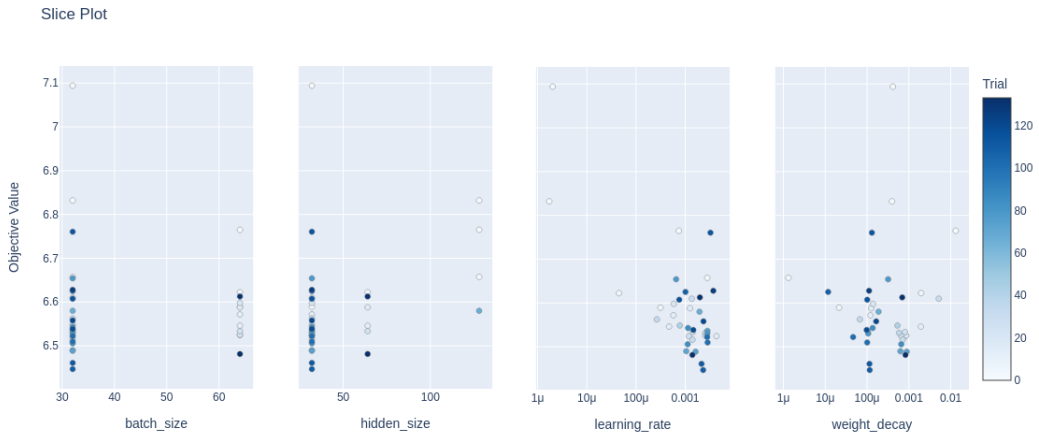


Figure 6.9: *Slice plot of the optimization results found through Optuna. From the 136 trials, 95 were pruned before completion and 39 were completed. The best trial achieved a loss value of 6.446797407 with the following hyperparameter values: batch size= 32, hidden size= 32, learning rate= 0.00231519996, weight decay= 0.0001155681898*

Chapter 7

Results and experiments

In this chapter we will describe the experiments we performed to compare the different building blocks for our model that we described in Chapter 6. To better understand the options available to build the best model possible, we begin by analyzing the proposed embeddings, first we compare the transformer based embeddings and then add GloVe to the analysis. With the "best" embedding selected we compare both proposed architectures and then lastly we train our definitive model based on the results of the previous experiments.

7.1 Comparing transformer based embeddings

To compare the transformer embeddings, we trained 7 pointer networks to compute the loss and accuracy graphs for these models in the validation and training datasets. In general, after the fourth epoch of training we began to see signs of overfitting in all of our models so here we only present training results up to the fourth epoch. In Fig 7.2 we can see the L1 loss plots of training and validation. In both cases "paraphrase-albert-small-v2" performed significantly worse than other models at all times. The best performing model isn't as clean cut as the worst but "distiluse-base-multilingual-cased-v1" — or dbmc1 as we will call it from now on — was able to attain the lowest validation loss at the third epoch.

Alongside the L1 loss we recorded the jaccard binary accuracy score of the predictions at each epoch, which can be seen in Fig. 7.1 as the train and validation accuracy. Once again "paraphrase-albert-small-v2" was the worst performing model and dbmc1 was the best performing model.

Due to the unexpectedly great performance of dbmc1 we decided to test "distiluse-base-multilingual-cased-v2" to validate if we were correct in our initial assumption that including another 35 languages in the training process wouldn't bring us much benefit. In Fig. 7.3 and Fig. 7.4 we can see the results of their comparison. Against our expectations, "distiluse-base-multilingual-cased-v2" was capable of obtaining a lower validation loss than its predecessor at the third epoch, however its accuracy was consistently lower in both training and validation. Since accuracy should better translate into real world performance

Embedding	Acc train	Acc val	Loss train	Loss val
distiluse-base-multilingual-cased-v1	0.7296	0.7263	5.485	6.423
all-mpnet-base-v2	0.7215	0.7172	5.596	6.407
all-distilroberta-v1	0.724	0.7178	5.554	6.442
all-MiniLM-L12-v2	0.7234	0.7121	5.549	6.494
paraphrase-albert-small-v2	0.709	0.6961	5.747	6.559
paraphrase-multilingual-MiniLM-L12-v2	0.7242	0.7102	5.546	6.496
paraphrase-MiniLM-L3-v2	0.7245	0.714	5.51	6.449
distiluse-base-multilingual-cased-v2	0.7271	0.7144	5.528	6.437

Table 7.1: *Accuracy and Loss of the final epoch for the eight different transformer based embeddings. Amongst the eight models "distiluse-base-multilingual-cased-v2" achieved the lowest loss validation score at the third epoch with a loss value of 6.334 followed by dbmc1 with a loss validation of 6.374 also at the third epoch.*

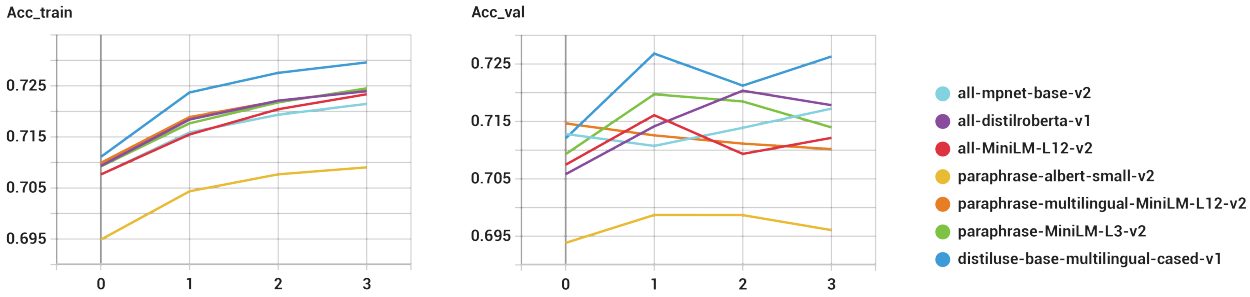


Figure 7.1: *Train and validation binary accuracy score of the seven transformer based models.*

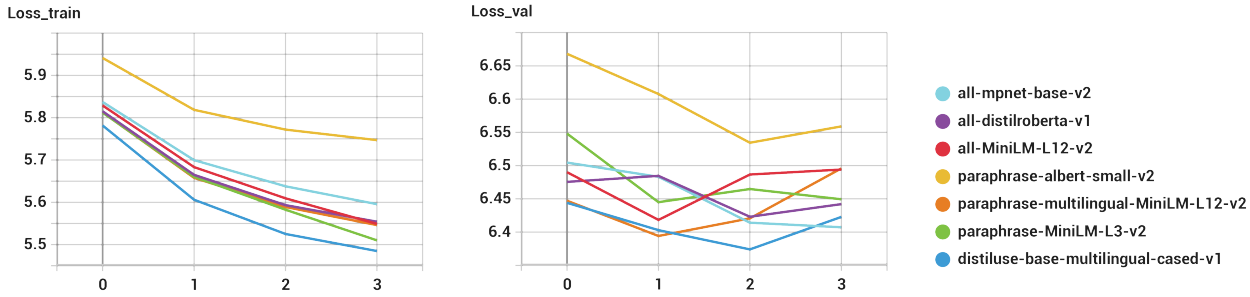


Figure 7.2: *L1 training and validation loss of the seven transformer based models.*

we will be keeping dbmc1 as the best performing model overall with a validation accuracy of 72.63%. Table 7.1 compiles the loss and accuracy results of these models.

7.2 Adding GloVe to the comparison

In Fig. 7.6 and Fig. 7.5 we compare the pointer network models trained with transformer based embeddings with their GloVe counterpart, which was the best performing model we obtained through Optuna in section 6.4. In the interest of clarity we decided to omit some of the transformer models from the graphs to avoid clutter, we plot only the best

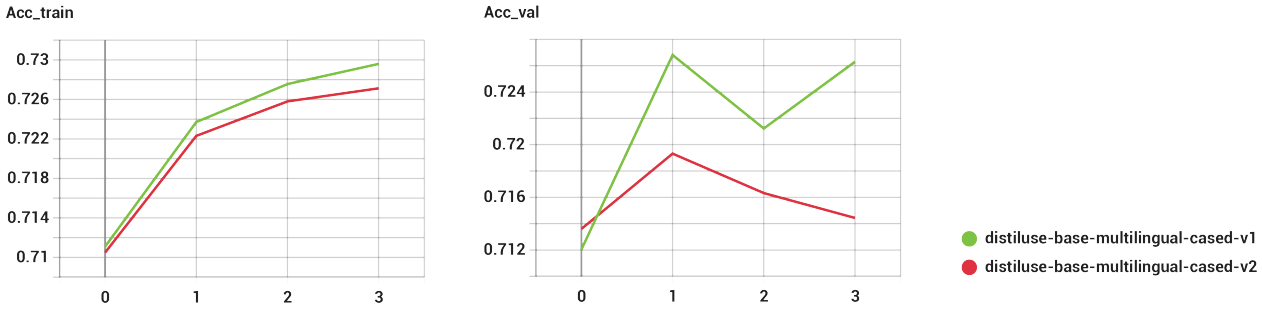


Figure 7.3: *Comparison of accuracy in validation and train sets between models dbmc1 and "distiluse-base-multilingual-cased-v2".*

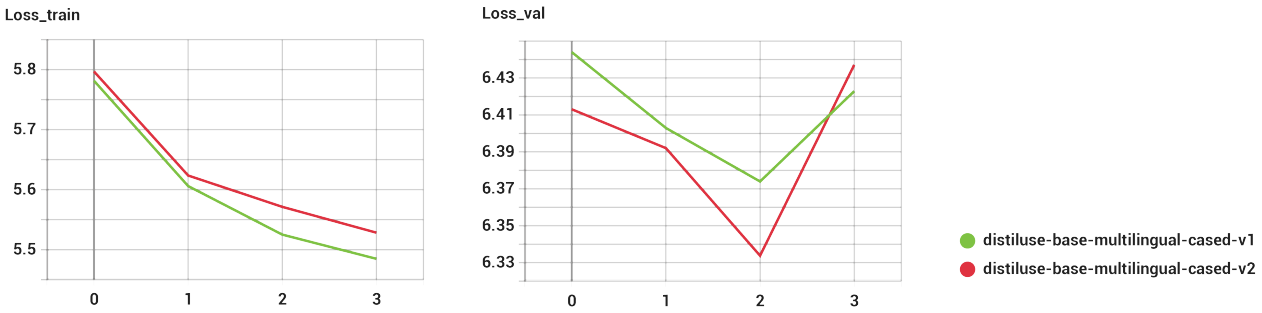


Figure 7.4: *Comparison of loss in validation and train sets between models dbmc1 and "distiluse-base-multilingual-cased-v2".*

performing transformer-based model and the two other that are adjacent in performance to the GloVe-based model. The graph makes it clear that the GloVe-based model performed consistently worse than all the transformer-based models, except for model "paraphrase-albert-small-v2" which was the worst performing one. Table 7.2 compiles the final accuracy and loss of all of the presented transformer models and the GloVe model.

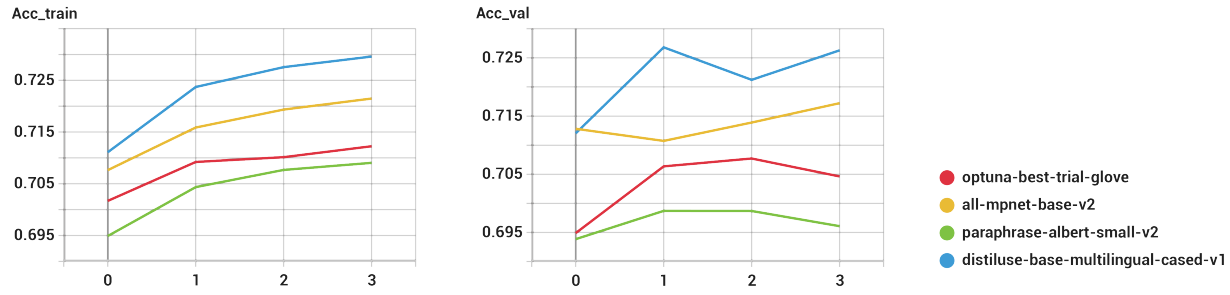


Figure 7.5: *Plots comparing the training and validation accuracy of four pointer networks trained with a GloVe embedding and 3 other transformer based embedding.*

7.3 Comparing architectures

In Fig. 7.7 and Fig. 7.8 we compare our two architectures, the pointer network and our so-called "simple RNN". We trained both models with the best performing embedding we

Embedding	Acc train	Acc val	Loss train	Loss val
distiluse-base-multilingual-cased-v1	0.7296	0.7263	5.485	6.423
all-mpnet-base-v2	0.7215	0.7172	5.596	6.407
all-distilroberta-v1	0.724	0.7178	5.554	6.442
all-MiniLM-L12-v2	0.7234	0.7121	5.549	6.494
paraphrase-albert-small-v2	0.709	0.6961	5.747	6.559
paraphrase-multilingual-MiniLM-L12-v2	0.7242	0.7102	5.546	6.496
paraphrase-MiniLM-L3-v2	0.7245	0.714	5.51	6.449
distiluse-base-multilingual-cased-v2	0.7271	0.7144	5.528	6.437
glove.840B.300d	0.7123	0.7046	5.64	6.447

Table 7.2: *Accuracy and Loss of the final epoch for the eight different transformer based embeddings and the GloVe based embedding.*

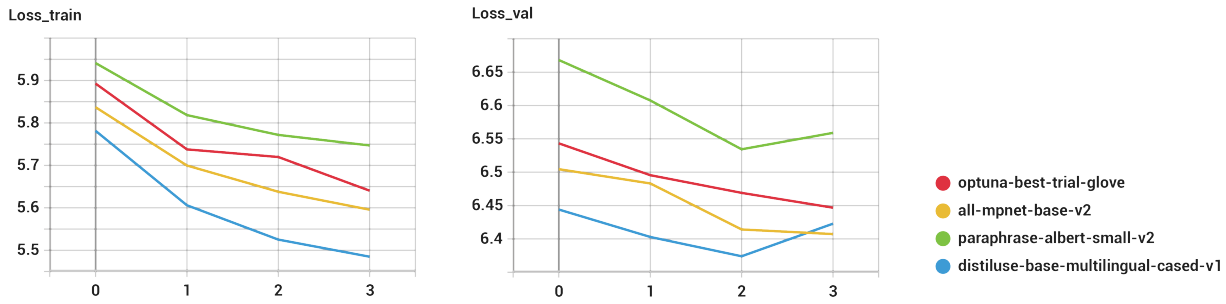


Figure 7.6: *Plots comparing the training and validation loss of four pointer networks trained with a GloVe embedding and 3 other transformer based embedding.*

found in previous sections, the dbmc1 embedding, and found that the Ptr-Net outperformed the simple RNN in all 4 metrics as can be seen in Table 7.3.

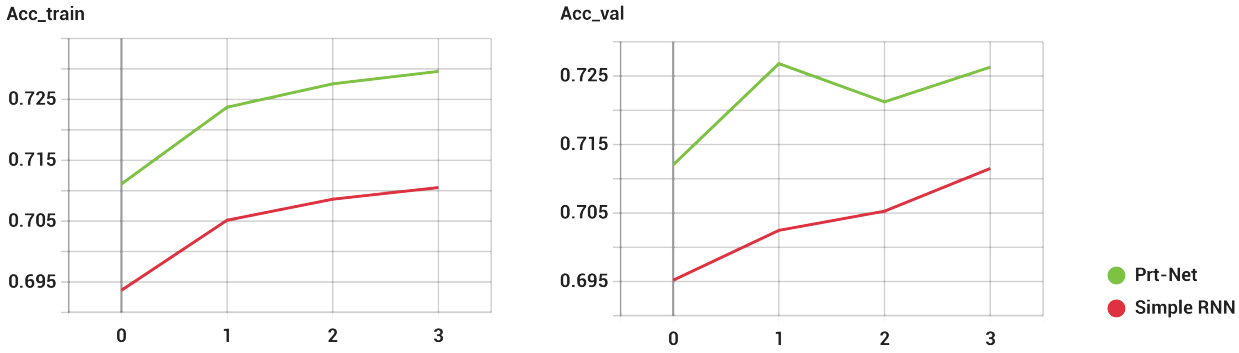


Figure 7.7: *Plots comparing the training and validation accuracy of our two architectures trained with the dbmc1 embedding.*

7.4 Best Model

Taking into account all our previous findings we can conclude that the pointer network architecture combined, with the dbmc1 embedding and the hyperparameters found by

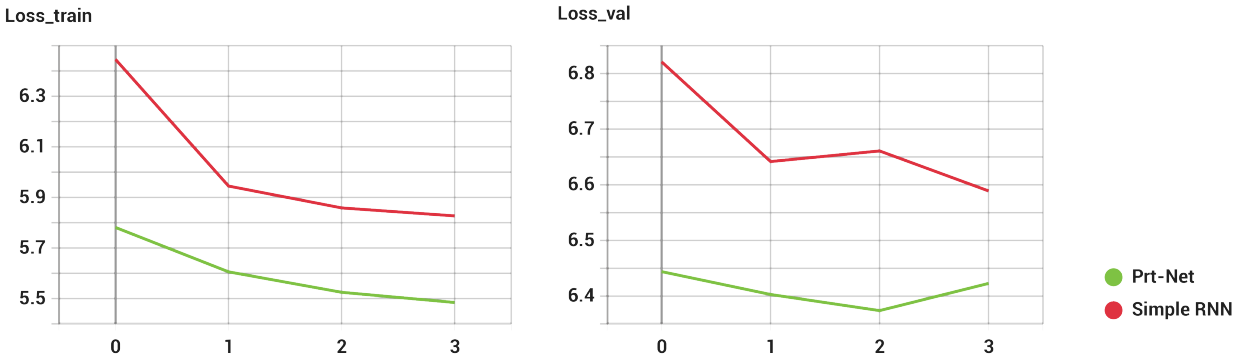


Figure 7.8: *Plots comparing the training and validation loss of our two architectures trained with the dbmc1 embedding.*

Architecture	Acc train	Acc val	Loss train	Loss val
Ptr-Net	0.7296	0.7263	5.485	6.423
Simple RNN	0.7105	0.7115	5.827	6.589

Table 7.3: *Table of the final (fourth epoch) loss and accuracy in the training and validation sets for our two different architectures.*

Optuna form our best performing model, Fig. 7.9 illustrates succinctly this final model. In Fig. 7.10 and Fig. 7.11 we present the accuracy and loss plots for this model in the training and testing datasets and in Table 7.4 we compile its final accuracy and loss scores.

Architecture	Acc train	Acc test	Loss train	Loss test
Ptr-Net and distiluse-base-multilingual-cased-v1 (Best Model)	0.7274	0.7275	5.529	5.768

Table 7.4: *Accuracy and loss of the final epoch in the training and test sets for the final model, trained with the Ptr-Net architecture and the dbmc1 embedding.*

As an alternative, dbmc1 could be swapped by the GloVe embedding in the interest of speeding up inference time. Albeit when doing a single prediction the inference time of dbmc1 is greater than that of GloVe it is not as significant of an increase as when training the models where the process takes multiple days for each transformer based embedding over the few hours of GloVe based models. This decision should be taken case by case, for some applications keeping the inference time as small as possible may be more important than the 0.01% performance gain of using dbmc1.

Whatever choice is made in a hypothetical deployment aside, we believe this model achieved the objectives we set. In conjunction with a language server it can easily execute function extraction refactorings and in conjunction with a refactoring opportunity detector these 3 components together could become a IDE plugin that automates the entirety of the function extraction process.

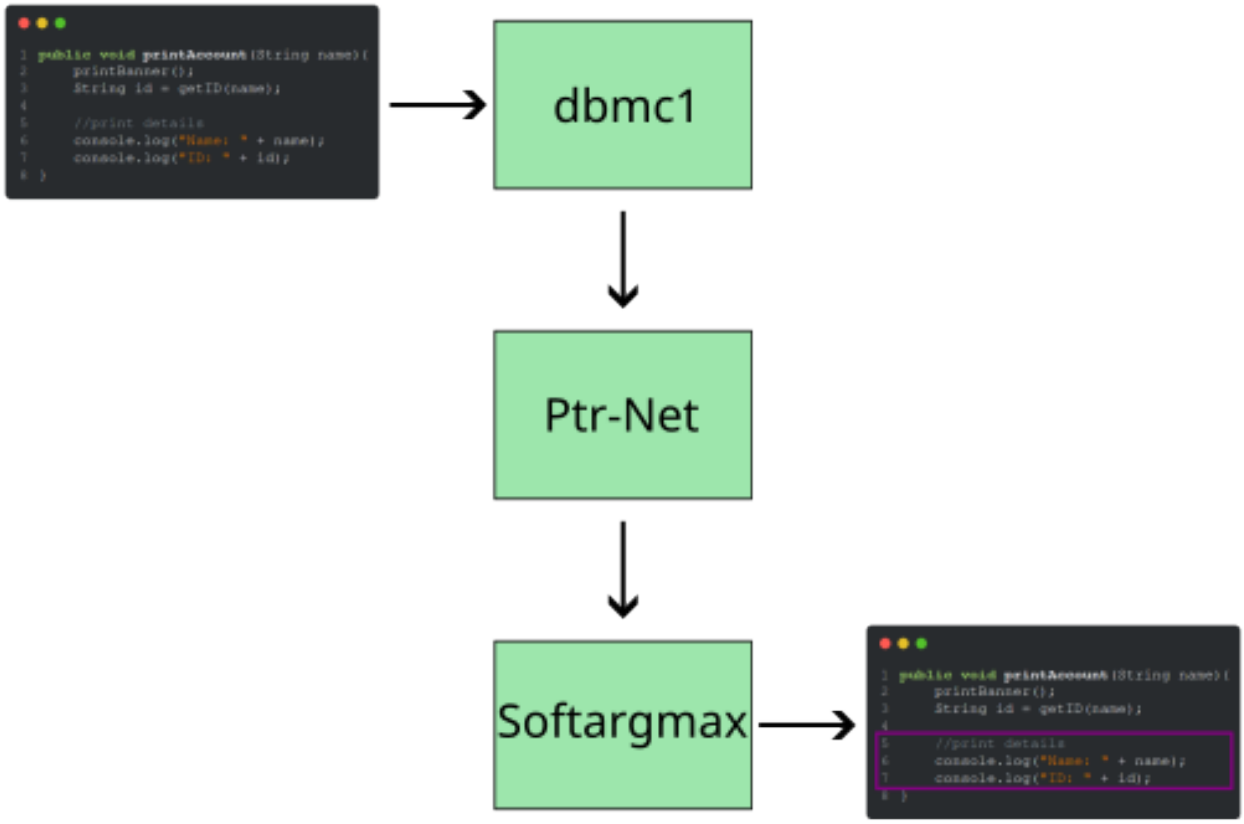


Figure 7.9: *Illustration of our final and best performing model. As previously mentioned, the hyper-parameters were chosen based on our Optuna experiments.*

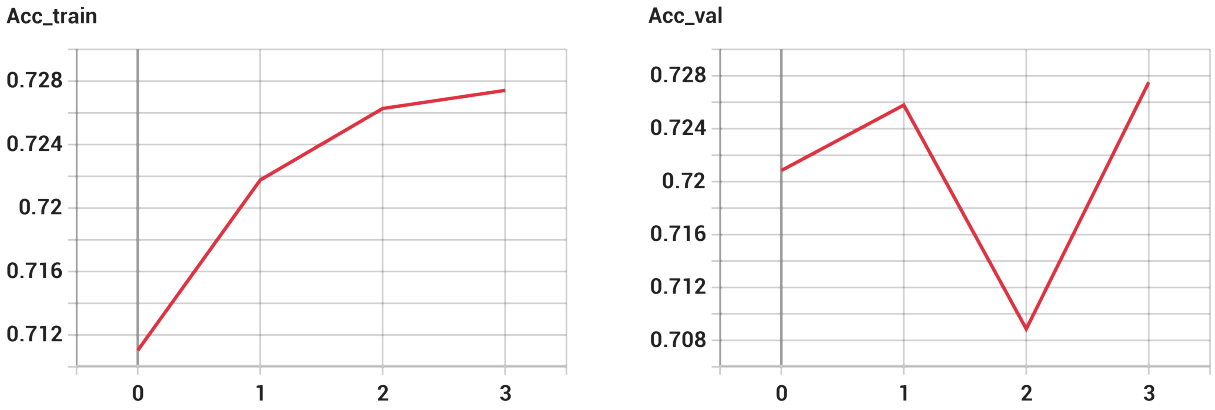


Figure 7.10: *Training and testing plots of accuracy of our best performing model in validation. Final accuracy value: 0.7275.*

7.4.1 Publication of results

The student is preparing to submit these results as a scientific paper to a peer reviewed journal.

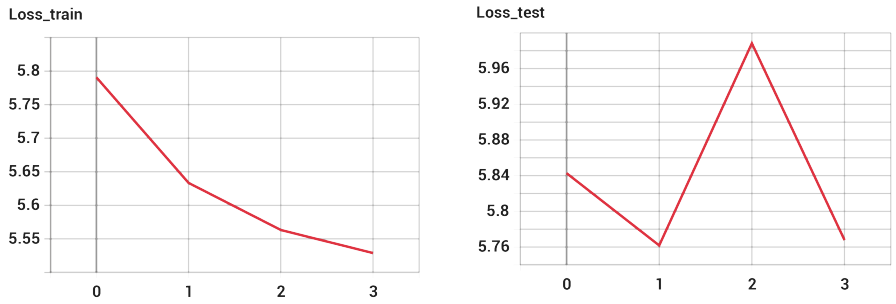


Figure 7.11: *Training and testing plots of loss of our best performing model in validation. Final loss value: 5.768.*

Chapter 8

Conclusion

In conclusion we believe we were able to achieve both of the goals set at the beginning of this project, our results show that is indeed possible for a deep learning model to predict fine-grained refactorings and we were successful in creating a number of models for automated function extraction refactoring.

Our final model consisted of the pointer network architecture the "distiluse-base-multilingual-cased-v1" embedding and the hyperparameter values obtained with Optuna, with this model we achieved a test loss and a test accuracy of 5.768 and 0.7275 respectively. We believe this results once again corroborate the hypothesis of naturality and show the soundness of our approach.

However, the models we trained were always hovering an accuracy of around 70%. Changing architectures, embeddings and hyperparameter values had a clear effect in the performance of the models but always in an incremental fashion with small gains to performance.

We hypothesize that to achieve significant gains in performance we would need to explore embeddings that also leverage the source code itself (by, for example, utilizing abstract syntax trees) instead of solely relying on natural language models. This is a hypothesis that we are interested in pursuing in future work.

We were also able to develop a new Java code refactoring dataset for function extractions that was over 60% bigger than the biggest dataset of its type published at the moment of elaboration of this report.

Through this we also hope to show that deep learning models are capable of predicting fine-grained refactorings, that they are able to dictate how exactly a snippet of code should be altered to obtain a successful refactoring.

Appendix A

AST Printer

The code utilized to print the AST present in Fig. 4.2 can be seen in Listing 2.

```
import com.github.javaparser.*;
import com.github.javaparser.ast.*;
import com.github.javaparser.printer.*;

import java.io.*;
import java.util.*;

public class ast_printer {

    public static void main(String[] args) throws Exception {
        String file_path = args[0];
        CompilationUnit cu = StaticJavaParser.parse(new File(file_path));
        DotPrinter printer = new DotPrinter(true);
        try (FileWriter fileWriter = new FileWriter(file_path + "_ast.dot");
                PrintWriter printWriter = new PrintWriter(fileWriter)) {
            printWriter.print(printer.output(cu));
        }
    }
}
```

Listing 2: *Code utilized to print Java ASTs using the `JavaParser` package.*

Appendix B

Data Scrapping Source Code

In the interest of reproducibility and a higher scientific standard, we provide the source code utilized to generate the dataset described in this work and most of its results.

```
#!/bin/bash
filename='repos.txt'

while read line; do
# reading each line

git clone $line

done < $filename

echo 'finished'
```

Listing 3: *Small bash script used to clone all the repositories listed in 'repos.txt'.*

```bash
#!/bin/bash
search_dir="cloned_repos"

function mine_repo(){
  repo=$1
  repo=$(basename "$repo")
  if [[ ! -f "./jsons/$repo.json" ]]; then
    echo "mining $repo"
    ./RefactoringMiner-2.1.0/bin/RefactoringMiner -a
     ↪  ./repos_clonados/$repo -json ./jsons/$repo.json
  fi
}
export -f mine_repo


echo "Starting mining"
ls $search_dir | parallel mine_repo
echo "Everything mined - the end - acabou - finito"
```

Listing 4: *Small bash script used to parallelize the mining of refactorings in all the repositories previously cloned.*

```python
data_path="./jsons/"
import json
import subprocess
import glob
import pandas as pd

from tqdm.asyncio import tqdm
from asyncio import run
from sqlalchemy.orm import sessionmaker, declarative_base
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy import Column, Integer, Text, String, Boolean

import logging

logger = logging.getLogger("processing_jsons")
logger.setLevel(logging.DEBUG)
f_handler = logging.FileHandler('processing_jsons.log')
f_handler.setFormatter(logging.Formatter('%(name)s - %(levelname)s -
 ↪  %(message)s'))
logger.addHandler(f_handler)

def process_json(json_name):

    results = []
```

```
24      with open(f"{data_path}{json_name}", "r") as read_file:
25          try:
26              json_payload = json.load(read_file)
27          except json.JSONDecodeError as e:
28              logger.debug(f"Broken json - {json_name}")
29              return
30
31          if not json_payload["commits"]:  #if is empty
32              logger.debug(f"Empty json - {json_name}")
33              return
34
35          repository = json_payload["commits"][0]['repository']
36
37          for commit in json_payload["commits"]:
38              for refactoring in commit["refactorings"]:
39                  if refactoring["type"] == "Extract Method":
40                      refactored_function =
                         ↪  refactoring["leftSideLocations"][0]
41                      start = refactored_function['startLine']
42                      end = refactored_function['endLine']
43
44                      extracted_lines = get_lines(
45                          refactoring["leftSideLocations"])
46
47                      #if for some reason the extracted lines are not
                         ↪  being encompassed by the function,
48                      #skip this refactoring
49                      if not all([start <= i <= end for i in
                         ↪  extracted_lines]):
50                          logger.debug(f"extracted lines are not being
                             ↪  encompassed by the function -
                             ↪  {json_name}")
51                          continue
52
53                      #if the body returns the empty string, lets skip
                         ↪  this refactoring
54                      #because we had a decoding error
55                      if not (refactored_function_body :=
                         ↪  get_function_body(
56                              start, end, repository, commit['sha1'],
57                              refactored_function['filePath'])):
58                          continue
59
60                      continuous_refac = check_continuous(
61                          start, extracted_lines,
                             ↪  refactored_function_body)
```

```
62
63                      results.append({
64                          'repository':
65                          repository,
66                          'sha1':
67                          commit['sha1'],
68                          'url':
69                          commit['url'],
70                          #transform into string to insert it in the
                             ↪  sqlite db
71                          "extracted_lines":
72                          str(extracted_lines),
73                          "shifted_extracted_lines":
74                          str([i - start for i in extracted_lines]),
75                          'refactoring_description':
76                          refactoring['description'],
77                          'file_path':
78                          refactored_function['filePath'],
79                          'func_startline':
80                          start,
81                          'func_endline':
82                          end,
83                          "shifted_extracted_lines_start":
84                          extracted_lines[0]-start,
85                          "shifted_extracted_lines_end":
86                          extracted_lines[-1]-start,
87                          'refactored_function_body':
88                          refactored_function_body,
89                          'continuous':
90                          continuous_refac
91                      })
92      return results
93
94
95  def get_lines(elements):
96      """
97      gets the specific line numbers of the lines extracted in the
         ↪  refactoring and return them as a sorted list.
98      """
99      lines = set()
100     for element in elements[1:]:
101         if element['startLine'] == element['endLine']:
102             lines.add(element['startLine'])
103         else:
104             lines.update(range(element['startLine'],
                 ↪  element['endLine'] + 1))
```

```
105
106     lines_list = list(lines)
107     lines_list.sort()
108     return lines_list
109
110
111 def get_function_body(start,
112                       end,
113                       repo,
114                       commit_hash,
115                       file_path,
116                       path_gits="/disk1/barzilay/repos_clonados"):
117     """
118     use git to get the file and then extract only the function body
119     """
120     repo = repo.split("/")[-1]
121     cmd = subprocess.run(["git", "show",
        ↪  f"{commit_hash}^:{file_path}"],
122                          cwd=f"{path_gits}/{repo}",
123                          capture_output=True)
124
125     try:
126         file_contents = cmd.stdout.decode("utf-8")
127     except UnicodeDecodeError as e:
128         logger.error(f"Decoding error at repo: {repo}", exc_info=True)
129         return ""
130     # java parser counts lines starting from 1 but python lists start
        ↪  at 0
131     body = "\n".join(file_contents.split("\n")[start - 1:end])
132     return body
133
134
135 def check_continuous(start, lines, body):
136     """
137     check if the function extraction was continuous or if it is
        ↪  composed of multiple line spans. Comments and blank lines are
        ↪  treated as part of refactorings, i.e. if the lines of code
        ↪  being extracted are continuous with the exception of comments
        ↪  in the middle of said lines, the refactoring will be
        ↪  considered continuous.
138     """
139     interval = {i for i in range(lines[0], lines[-1] + 1)}
140     difference = interval - set(lines)
141     to_check = [i - start for i in difference]
142     to_check.sort()
143
```

```
144         body_lines = body.split("\n")
145         long_comment = False
146 
147         for index in to_check:
148             line = body_lines[index].strip()
149 
150             #if whitespace or comment
151             if not line or line[0:2] == "//":
152                 continue
153 
154             elif line[0:2] == "/*" or long_comment == True:
155                 long_comment = True
156                 #check if the long comment ended in this line and if there
                     ↪  is any code after it
157                 end_long_comment = line.find("*/")
158                 if end_long_comment != -1:
159                     if end_long_comment + 2 != len(line): return False
160                     long_comment = False
161             else:
162                 return False
163 
164         return True
165 
166 
167 
168 Base = declarative_base()
169 
170 
171 class Refactoring(Base):
172         """
173         Refactoring class used for the ORM between our code base in SQLite
              ↪  and our python objects obtained after processing the JSON
              ↪  files.
174 
175         This class and its methods were developed with Rafael S. Durelli.
176         """
177         __tablename__ = 'refactoring'
178         id = Column(Integer, primary_key=True)
179         repository = Column(String(1000))
180         sha1 = Column(String(43))
181         url = Column(String(1000))
182         extracted_lines = Column(String(10000))
183         shifted_extracted_lines= Column(String(10000))
184         refactoring_description = Column(String(3000))
185         file_path = Column(String(1000))
186         func_startline = Column(Integer)
```

```
187     func_endline = Column(Integer)
188     continuous = Column(Boolean)
189     refactored_function_body = Column(Text())
190     shifted_extracted_lines_start= Column(Integer)
191     shifted_extracted_lines_end= Column(Integer)
192
193
194
195
196     def __repr__(self):
197         return f'<Refactoring> {self.id} {self.repository} {self.sha1}
            ↪ {self.url} {self.extracted_lines}
            ↪ {self.refactoring_description} {self.file_path}
            ↪ {self.func_startline} {self.func_endline}
            ↪ {self.refactored_function_body} '
198
199     @staticmethod
200     async def insert_refactoring_list(session, refactoring_list):
201         """
202         static method used to add a list of Refactoring python objects
            ↪  into the SQLite database in an asynchronous fashion.
203         """
204         async with session() as s:
205             s.add_all([
206                 Refactoring(**refactoring_to_insert)
207                 for refactoring_to_insert in refactoring_list
208             ])
209             await s.commit()
210
211
212 db_url = 'sqlite+aiosqlite:///refactorings.db'
213 engine = create_async_engine(db_url)
214
215 session = sessionmaker(engine,
216                        expire_on_commit=False,
217                        future=True,
218                        class_=AsyncSession)
219
220
221 async def create_database():
222     async with engine.begin() as conn:
223         await conn.run_sync(Base.metadata.drop_all)
224         await conn.run_sync(Base.metadata.create_all)
225
226
227
```

```
228 run(create_database())
229
230 jsons_list= glob.glob("*.json", root_dir=data_path)
231
232 with tqdm(jsons_list) as pbar:
233     async for json_file in pbar:
234         mined_refactorings=process_json(json_file)
235         if mined_refactorings:
236             await Refactoring.insert_refactoring_list(session,
                ↪  mined_refactorings)
237
```

Listing 5: *Python script used to process the JSON files into an actionable SQLite database of function extraction refactorings and their metadata.*

Appendix C

β impact on *softargmax*

Normally this discussion would be a part of the main thesis, however we feel that it lacks in rigor to convince skeptic readers but simply discarding it would be omission on our part. We wished to replicate this experiment in a matter to have statistical significance to demonstrate our findings but due to time constraints and the somewhat out-of-scope aspect of this experiment, training thousands of models for this was not feasible. So here we present a simple snapshot of what trying to overfit a model over a reduced dataset while varying the value of β would look like, Fig. C.1 presents the training loss plot.

While we were performing this experiment we realized that there was too much variability between runs with a same β value to be able to precisely describe its impact. However, some behaviors were found to be consistent: no model with a β value of 1000 or above was able to overfit the data, i.e. they were incapable of learning with an essentially fixed loss across epochs. Higher values of β lead to a better approximation of the argmax function but when β is to big it also starts approximating its discontinuity leading to a broken gradient flow and making the model ability to learn to come to a halt. To better understand this phenomenon and β impact on learning more experiments to analyze the spread of the loss function would be necessary.

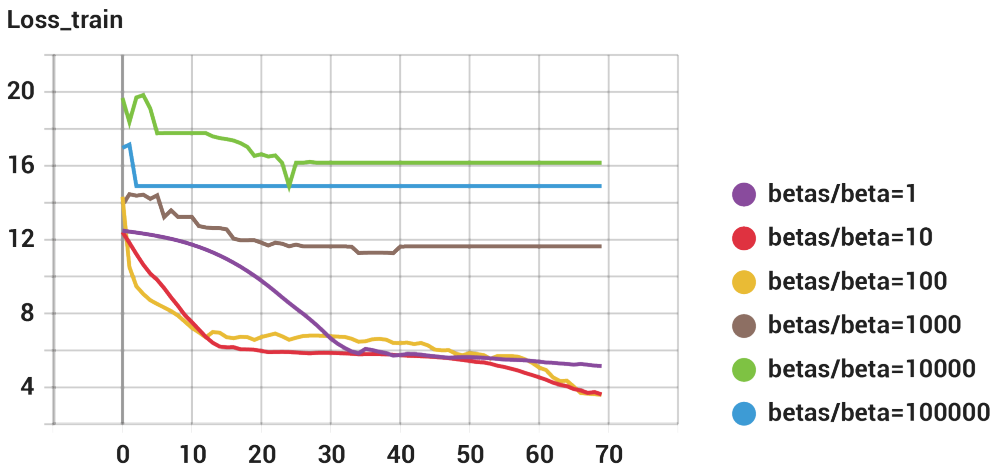


Figure C.1: *Training loss plot over the training epochs of a pointer network model for varying β values. Training was done over only 10 data points in order o explore the ability of the model to overfit, the initial rational was that if a model cannot even overfit a minuscule dataset it is not suitable for training with the entire dataset or that it may even be be broken.*

References

[Abid *et al.* 2020] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. *30 Years of Software Refactoring Research:A Systematic Literature Review*. 2020. arXiv: 2007.02194 [cs.SE] (cit. on p. 2).

[Adrian Rosebrock 2016] Adrian Rosebrock. *A visual equation for Intersection over Union (Jaccard Index)*. pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/. Accessed: 2023-05-13. 2016 (cit. on pp. xiii, 55).

[Aho *et al.* 2007] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, & Tools Second Edition*. 2007 (cit. on p. 34).

[Alighieri 130-] Dante Alighieri. *La divina commedia*. 130- (cit. on p. 7).

[Allamanis, Earl T Barr, *et al.* 2018] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. "A survey of machine learning for big code and naturalness". *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–37 (cit. on pp. 6, 7, 49).

[Allamanis, Earl T. Barr, *et al.* 2014] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. "Learning natural coding conventions". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014. doi: 10.1145/2635868.2635883. url: https://doi.org/10.1145%5C%2F2635868.2635883 (cit. on p. 6).

[Alon, Brody, *et al.* 2018] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. "Code2seq: generating sequences from structured representations of code". *arXiv preprint arXiv:1808.01400* (2018) (cit. on pp. 8, 37, 43).

[Alon, Zilberstein, *et al.* 2018] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. *code2vec: Learning Distributed Representations of Code*. 2018. arXiv: 1803.09473 [cs.LG] (cit. on pp. xi, 8, 37, 43).

[Amazon 2023] Amazon. *Amazon CodeWhisperer*. aws.amazon.com/codewhisperer/. Accessed: 2023-05-13. 2023 (cit. on p. 38).

[Aniche *et al.* 2020] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. "The effectiveness of supervised machine learning algorithms in predicting software refactoring". *arXiv preprint arXiv:2001.03338* (2020) (cit. on pp. xv, 33, 39–41, 43, 47).

[Apache Software Foundation 2023] Apache Software Foundation. *Apache Projects by language*. projects.apache.org/projects.html?language. Accessed: 2023-05-13. 2023 (cit. on p. 40).

[Appel 2004] Andrew W. Appel. *Modern Compiler Implementation in Java Second Edition*. 2004 (cit. on p. 34).

[Bahdanau *et al.* 2014a] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2014. doi: 10.48550/ARXIV.1409.0473. url: https://arxiv.org/abs/1409.0473 (cit. on p. 53).

[Bahdanau *et al.* 2014b] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". *arXiv preprint arXiv:1409.0473* (2014) (cit. on pp. xi, 25).

[Baker 2016] Monya Baker. "1,500 scientists lift the lid on reproducibility". *Nature* 533.7604 (2016) (cit. on p. 55).

[Ben Olmstead 1998] Ben Olmstead. *Malbolge*. esolangs.org/wiki/Malbolge. Accessed: 2023-05-13. 1998 (cit. on p. 7).

[Bergstra *et al.* 2011] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for hyper-parameter optimization". *Advances in neural information processing systems* 24 (2011) (cit. on p. 56).

[Bird *et al.* 2011] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. "Don't touch my code! examining the effects of ownership on software quality". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 4–14 (cit. on p. 39).

[Bulbapedia 2005] Bulbapedia. *Bulbapedia, the community-driven Pokemon encyclopedia*. bulbapedia.bulbagarden.net/wiki/Main_Page. Accessed: 2023-01-15. 2005 (cit. on pp. x, 20).

[Callison-Burch *et al.* 2006] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. "Re-evaluating the role of Bleu in machine translation research". In: *11th Conference of the European Chapter of the Association for Computational Linguistics*. Trento, Italy: Association for Computational Linguistics, Apr. 2006. url: https://www.aclweb.org/anthology/E06-1032 (cit. on p. 31).

[Chidamber and Kemerer 1994] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493 (cit. on p. 14).

[Cho *et al.* 2014] Kyunghyun Cho *et al.* "Learning phrase representations using rnn encoder-decoder for statistical machine translation". *arXiv preprint arXiv:1406.1078* (2014) (cit. on pp. 5, 24).

[Christopher Manning 2020] Christopher Manning. *Lecture 2: Word Vectors,Word Senses, and Classifier Review - CS224N/Ling284*. web.stanford.edu/class/archive/cs/cs224n/cs224n.1204/slides/cs224n-2020-lecture02-wordvecs2.pdf. Accessed: 2023-01-15. 2020 (cit. on pp. x, 23).

[Christopher Olah 2015] Christopher Olah. *Understanding LSTM Networks*. colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 2023-05-26. 2015 (cit. on pp. x, 23).

[Coleman *et al.* 1994] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. "Using metrics to evaluate software system maintainability". *Computer* 27.8 (1994), pp. 44–49 (cit. on p. 15).

[Croft 2008] William Croft. "Evolutionary linguistics". *Annual review of anthropology* 37 (2008), pp. 219–234 (cit. on p. 4).

[Daniel *et al.* 2007] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. "Automated testing of refactoring engines". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 185–194. isbn: 9781595938114. doi: 10.1145/1287624.1287651. url: https://doi.org/10.1145/1287624.1287651 (cit. on p. 13).

[Devlin *et al.* 2018] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: pre-training of deep bidirectional transformers for language understanding". *arXiv preprint arXiv:1810.04805* (2018) (cit. on pp. 5, 29, 50).

[Eclipse Foundation 2021] Eclipse Foundation. *Eclipse JDT Language Server*. https://github.com/eclipse/eclipse.jdt.ls. Accessed: 2021-07-05. 2021 (cit. on p. 16).

[Esolang wiki 2023a] Esolang wiki. *brainfuck*. esolangs.org/wiki/Brainfuck. Accessed: 2023-05-13. 2023 (cit. on pp. ix, 8).

[Esolang wiki 2023b] Esolang wiki. *Malbolge*. esolangs.org/wiki/Malbolge. Accessed: 2023-05-13. 2023 (cit. on pp. ix, 8).

[F-droid 2023] F-droid. *F-droid Website*. f-droid.org/en/. Accessed: 2023-05-13. 2023 (cit. on p. 40).

[Firth 1957] John Firth. "A synopsis of linguistic theory, 1930-1955". *Studies in linguistic analysis* (1957), pp. 10–32 (cit. on p. 21).

[Fowler 1999] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. isbn: 0-201-48567-2 (cit. on pp. ix, 2, 12–14).

[Galassi *et al.* 2021] Andrea Galassi, Marco Lippi, and Paolo Torroni. "Attention in natural language processing". *IEEE Transactions on Neural Networks and Learning Systems* 32.10 (Oct. 2021), pp. 4291–4308. doi: 10.1109/tnnls.2020.3019893. url: https://doi.org/10.1109%5C%2Ftnnls.2020.3019893 (cit. on pp. xi, 26).

[Gibney 2022] Elizabeth Gibney. "Could machine learning fuel a reproducibility crisis in science?" *Nature* 608.7922 (2022), pp. 250–251. doi: 10.1038/d41586-022-02035-w. url: https://doi.org/10.1038/d41586-022-02035-w (cit. on p. 55).

[GitHub 2023] GitHub. *Introducing GitHub Copilot X*. github.com/features/preview/copilot-x. Accessed: 2023-05-13. 2023 (cit. on p. 38).

[GitHub Next 2023] GitHub Next. *Code Brushes*. githubnext.com/projects/code-brushes/. Accessed: 2023-05-13. 2023 (cit. on p. 38).

[gojp 2023] gojp. *Go Report Card*. https://github.com/gojp/goreportcard. Accessed: 2023-05-03. 2023 (cit. on p. 14).

[Golubev *et al.* 2021] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. "One thousand and one stories: a large-scale survey of software refactoring". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1303–1313 (cit. on pp. ix, 2).

[Grover and Leskovec 2016] Aditya Grover and Jure Leskovec. *node2vec: Scalable Feature Learning for Networks*. 2016. arXiv: 1607.00653 [cs.SI] (cit. on p. 21).

[Heil *et al.* 2021] Benjamin J Heil *et al.* "Reproducibility standards for machine learning in the life sciences". *Nature Methods* 18.10 (2021), pp. 1132–1135 (cit. on p. 55).

[Hellendoorn *et al.* 2015] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. "Will they like this? evaluating code contributions with language models". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 2015, pp. 157–167. doi: 10.1109/MSR.2015.22 (cit. on p. 6).

[Hindle *et al.* 2016] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. "On the naturalness of software". *Communications of the ACM* 59.5 (2016), pp. 122–131 (cit. on p. 7).

[Hochreiter and Schmidhuber 1997] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 22).

[C. F. Hockett and C. D. Hockett 1960] Charles F Hockett and Charles D Hockett. "The origin of speech". *Scientific American* 203.3 (1960), pp. 88–97 (cit. on p. 3).

[Hofstadter 1979] Douglas R Hofstadter. "Gödel, escher, bach: an eternal golden braid," (1979) (cit. on p. 5).

[Hope C. Dawson 2016] Michael Phelan Hope C. Dawson. *Language Files Materials for an Introduction to Language and Linguistics*. The Ohio State University Press, 2016 (cit. on pp. 3, 4).

[Hutson 2018] Matthew Hutson. *Artificial intelligence faces reproducibility crisis*. 2018 (cit. on p. 55).

[Izre'el 2003] Shlomo Izre'el. "The emergence of spoken israeli hebrew". *Corpus Linguistics and Modern Hebrew: Towards the Compilation of the Corpus of Spoken Israeli Hebrew* (2003), pp. 85–104 (cit. on p. 4).

[Jaccard 1912] Paul Jaccard. "The distribution of the flora in the alpine zone. 1". *New phytologist* 11.2 (1912), pp. 37–50 (cit. on p. 55).

[Jay Alammar 2019] Jay Alammar. *The Illustrated Word2vec*. jalammar.github.io/illustrated-word2vec/. Accessed: 2023-01-15. 2019 (cit. on pp. x, 22).

[Juan Orozco Villalobos 2020] Juan Orozco Villalobos. *One-hot encoding with Pokemon*. www.brainstobytes.com/one-hot-encoding-with-pokemon/. Accessed: 2023-01-15. 2020 (cit. on pp. x, 20).

[Kapoor and Narayanan 2022] Sayash Kapoor and Arvind Narayanan. *Leakage and the Reproducibility Crisis in ML-based Science*. 2022. arXiv: 2207.07048 [cs.LG] (cit. on p. 55).

[Kim *et al.* 2014] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. "An empirical study of refactoringchallenges and benefits at microsoft". *IEEE Transactions on Software Engineering* 40.7 (2014), pp. 633–649 (cit. on pp. 2, 13).

[Knuth 1984] Donald Ervin Knuth. "Literate programming". *The computer journal* 27.2 (1984), pp. 97–111 (cit. on p. 6).

[Lan *et al.* 2019] Zhenzhong Lan *et al. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2019. doi: 10.48550/ARXIV.1909.11942. url: https://arxiv.org/abs/1909.11942 (cit. on p. 50).

[Latent Space 2023] Latent Space. *Training a SOTA Code LLM in 1 week and Quantifying the Vibes - with Reza Shabani of Replit*. www.latent.space/p/reza-shabani. Accessed: 2023-05-26. 2023 (cit. on p. 7).

[Le and Mikolov 2014] Quoc V. Le and Tomás Mikolov. "Distributed representations of sentences and documents". *CoRR* abs/1405.4053 (2014). arXiv: 1405.4053. URL: http://arxiv.org/abs/1405.4053 (cit. on p. 21).

[Liu *et al.* 2019] Yinhan Liu *et al.* "Roberta: a robustly optimized bert pretraining approach". *arXiv preprint arXiv:1907.11692* (2019) (cit. on p. 50).

[Luong *et al.* 2015] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. *Effective Approaches to Attention-based Neural Machine Translation*. 2015. arXiv: 1508.04025 [cs.CL] (cit. on pp. xi, 27).

[Martin Fowler 2023a] Martin Fowler. *Catalog of Refactorings*. https://refactoring.com/catalog/. Accessed: 2023-01-15. 2023 (cit. on p. 13).

[Martin Fowler 2023b] Martin Fowler. *Catalog: Change Function Declaration*. refactoring.com/catalog/changeFunctionDeclaration.html. Accessed: 2023-05-13. 2023 (cit. on pp. xi, 33).

[Maurice 1977] H Halstead Maurice. *Elements of software science (operating and programming systems series)*. 1977 (cit. on p. 15).

[McCabe 1976] Thomas J McCabe. "A complexity measure". *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320 (cit. on p. 15).

[mensweardog 2023] mensweardog. *Menswear Dog - Bodhi, The Most Stylish Dog in the World*. www.instagram.com/mensweardog/. Accessed: 2023-05-26. 2023 (cit. on pp. xi, 26).

[Merriam-Webster, Inc. 2023] Merriam-Webster, Inc. *How many words are there in English?* www.merriam-webster.com/help/faq-how-many-english-words. Accessed: 2023-01-15. 2023 (cit. on p. 19).

[Microsoft 2016] Microsoft. *Language Server Protocol Specification*. https://microsoft.github.io/language-server-protocol/specifications/specification-current/. Accessed: 2021-07-05. 2016 (cit. on p. 16).

[Microsoft 2021] Microsoft. *Language Server Extension Guide*. https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. Accessed: 2021-07-05. 2021 (cit. on pp. ix, 16).

[Mikolov *et al.* 2013] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space". *arXiv preprint arXiv:1301.3781* (2013) (cit. on p. 21).

[Moghadam *et al.* 2021] Iman Hemati Moghadam, Mel Ó Cinnéide, Faezeh Zarepour, and Mohamad Aref Jahanmir. "Refdetect: a multi-language refactoring detection tool based on string alignment". *IEEE Access* 9 (2021), pp. 86698–86727 (cit. on pp. xv, 44, 45).

[Nils Reimers, Iryna Gurevych 2022a] Nils Reimers, Iryna Gurevych. *SBERT package*. https://www.sbert.net/. Accessed: 2022-11-22. 2022 (cit. on p. 51).

[Nils Reimers, Iryna Gurevych 2022b] Nils Reimers, Iryna Gurevych. *SBERT pretrained models*. https://www.sbert.net/docs/pretrained_models.html. Accessed: 2022-11-22. 2022 (cit. on pp. xii, 50, 51).

[Opdyke 1992] William F Opdyke. *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign, 1992 (cit. on p. 11).

[OpenAI 2023] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL] (cit. on p. 6).

[Optuna team 2022] Optuna team. *Optuna website*. https://optuna.org/. Accessed: 2022-11-22. 2022 (cit. on p. 56).

[Palomba *et al.* 2013] F. Palomba *et al.* "Detecting bad smells in source code using change history information". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 268–278. doi: 10.1109/ASE.2013.6693086 (cit. on p. 13).

[Papineni *et al.* 2002] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. "Bleu: a method for automatic evaluation of machine translation". In: *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics. 2002, pp. 311–318 (cit. on p. 31).

[Pat Hawks 2018] Pat Hawks. *An example parse tree*. en.wikipedia.org/wiki/File:Parse_Tree_1.svg. Accessed: 2023-05-26. 2018 (cit. on pp. ix, 5).

[Pennington *et al.* 2014] Jeffrey Pennington, Richard Socher, and Christopher D Manning. "Glove: global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543 (cit. on pp. 22, 51).

[Peters *et al.* 2018] Matthew E Peters *et al.* "Deep contextualized word representations". *arXiv preprint arXiv:1802.05365* (2018) (cit. on p. 5).

[Rajpurkar, Jia, *et al.* 2018] Pranav Rajpurkar, Robin Jia, and Percy Liang. *Know What You Don't Know: Unanswerable Questions for SQuAD*. 2018. arXiv: 1806.03822 [cs.CL] (cit. on p. 31).

[Rajpurkar, Zhang, *et al.* 2016] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. "Squad: 100,000+ questions for machine comprehension of text". *arXiv preprint arXiv:1606.05250* (2016) (cit. on pp. xi, 28, 29, 31).

[Randall Munroe 2006] Randall Munroe. *Computational Linguists*. xkcd.com/114/. Accessed: 2023-05-26. 2006.

[Refactoring.ai 2021] Refactoring.ai. *Data-Collection - Collect refactorings with metrics from java source code*. github.com/refactoring-ai/Data-Collection. Accessed: 2023-05-26. 2021 (cit. on pp. 43, 46).

[Reimers and Gurevych 2019] Nils Reimers and Iryna Gurevych. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. 2019. doi: 10.48550/ARXIV.1908.10084. url: https://arxiv.org/abs/1908.10084 (cit. on pp. 21, 50).

[Replit 2023] Replit. *Meet Ghostwriter, your partner in code*. replit.com/site/ghostwriter. Accessed: 2023-05-19. 2023 (cit. on p. 38).

[Rogers *et al.* 2020] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. "A primer in BERTology: what we know about how BERT works". *Transactions of the Association for Computational Linguistics* 8 (2020), pp. 842–866. doi: 10.1162/tacl_a_00349. url: https://aclanthology.org/2020.tacl-1.54 (cit. on p. 29).

[Sanh *et al.* 2019] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2019. doi: 10.48550/ARXIV.1910.01108. url: https://arxiv.org/abs/1910.01108 (cit. on p. 50).

[Silva *et al.* 2020] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. "Refdiff 2.0: a multi-language refactoring detection tool". *IEEE Transactions on Software Engineering* 47.12 (2020), pp. 2786–2802 (cit. on p. 44).

[Sipser 2013] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2013 (cit. on p. 5).

[Song *et al.* 2020] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. *MPNet: Masked and Permuted Pre-training for Language Understanding*. 2020. doi: 10.48550/ARXIV.2004.09297. url: https://arxiv.org/abs/2004.09297 (cit. on p. 50).

[Spolsky 1995] Bernard Spolsky. "Conditions for language revitalization: a com parison of the cases of hebrew and maori". *Current Issues in Language & Society* 2.3 (1995), pp. 177–201 (cit. on p. 4).

[Sutskever *et al.* 2014] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014, pp. 3104–3112 (cit. on p. 24).

[TabNine 2020] TabNine. *TabNine HomePage*. https://tabnine.com/. Accessed: 2020-01-15. 2020 (cit. on p. 38).

[Tensor Flow 2020] Tensor Flow. *Word representation tutorial*. https://www.tensorflow.org/tutorials/representation/word2vec. Accessed: 2020-01-15. 2020 (cit. on pp. x, 22).

[The Charity Development Group 1996] The Charity Development Group. *The CHARITY Home Page*. pll.cpsc.ucalgary.ca/charity1/www/home.html. Accessed: 2023-05-13. 1996 (cit. on p. 5).

[Thomas Schoch 2006a] Thomas Schoch. *Piet - a language where the programs are works of modern art*. www.dangermouse.net/esoteric/piet.html. Accessed: 2023-05-13. 2006 (cit. on p. 7).

[Thomas Schoch 2006b] Thomas Schoch. *Piet-program printing "Piet", codel size 6*. retas.de/thomas/computer/programs/useless/piet/Piet/index.html. Accessed: 2023-05-13. 2006 (cit. on pp. ix, 8).

[Touvron *et al.* 2023] Hugo Touvron *et al. LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL] (cit. on p. 6).

[Tsantalis, Chaikalis, *et al.* 2008] Nikolaos Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. "Jdeodorant: identification and removal of type-checking bad smells". In: *2008 12th European Conference on Software Maintenance and Reengineering*. 2008, pp. 329–331. doi: 10.1109/CSMR.2008.4493342 (cit. on p. 13).

[Tsantalis, Ketkar, *et al.* 2020] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. "Refactoringminer 2.0". *IEEE Transactions on Software Engineering* 48.3 (2020), pp. 930–950 (cit. on pp. xii, xv, 40, 43–45).

[Tsantalis, Mansouri, *et al.* 2018] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. "Accurate and efficient refactoring detection in commit history". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 483–494 (cit. on p. 44).

[Tufano *et al.* 2015] M. Tufano *et al.* "When and why your code starts to smell bad". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 403–414. doi: 10.1109/ICSE.2015.59 (cit. on p. 13).

[Unger *et al.* 2018] Mohse Unger, Bracha Shapira, Lior Rokach, and Amit Livne. "Inferring contextual preferences using deep encoder-decoder learners". *New Review of Hypermedia and Multimedia* (Sept. 2018). doi: 10.1080/13614568.2018.1524934 (cit. on pp. x, 25).

[Urban Müller 1993] Urban Müller. *brainfuck*. esolangs.org/wiki/Brainfuck. Accessed: 2023-05-13. 1993 (cit. on p. 7).

[Vaswani *et al.* 2017] Ashish Vaswani *et al.* "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008 (cit. on pp. xi, 5, 28, 30).

[Vinyals *et al.* 2015] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer networks". *arXiv preprint arXiv:1506.03134* (2015) (cit. on pp. xi, 28, 53).

[S. Wang and Jiang 2016] Shuohang Wang and Jing Jiang. "Machine comprehension using match-lstm and answer pointer". *arXiv preprint arXiv:1608.07905* (2016) (cit. on p. 28).

[W. Wang *et al.* 2020] Wenhui Wang *et al. MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers*. 2020. doi: 10.48550/ARXIV.2002.10957. url: https://arxiv.org/abs/2002.10957 (cit. on p. 50).

[Weng 2018] Lilian Weng. "Attention? attention!" *lilianweng.github.io* (2018). url: https://lilianweng.github.io/posts/2018-06-24-attention/ (cit. on pp. x, 24).

[Wildenhain 2017] Tom Wildenhain. "On the turing completeness of ms powerpoint". In: *The Official Proceedings of the Eleventh Annual Intercalary Workshop about Symposium on Robot Dance Party in Celebration of Harry Q Bovik's*. Vol. 2. 2017, pp. 102–106 (cit. on p. 5).

[Zügner *et al.* 2021] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. *Language-Agnostic Representation Learning of Source Code from Structure and Context*. 2021. arXiv: 2103.11318 [cs.LG] (cit. on pp. xii, 37, 38, 43).