INAUGURAL – DISSERTATION

zur

Erlangung der Doktorwürde

der

Gesamtfakultät für Mathematik, Ingenieur- und Naturwissenschaften

der

Ruprecht – Karls – Universität

Heidelberg

vorgelegt von

Le, Thi Kim Tuyen, M.Sc.

aus Ben Tre, Vietnam

Tag der mündlichen Prüfung: .....

Printed and/or published with the support of
the German Academic Exchange Service (DAAD)

Accelerated Programming for Data Analysis and Processing

Betreuer: Prof. Dr. Artur Andrzejak Heidelberg University

Abstract

In the past decade, data science has made remarkable progress, evidenced by the proliferation of data-driven strategies, the rapid growth of data science-related jobs, and the expansion of university curricula in this field. Consequently, data strengthens its role as a paramount asset for organizations. Nonetheless, along with great benefits come nontrivial challenges.

Particularly, (1) many domain experts, who are proficient in their respective fields but lack programming skills, face difficulties in learning and utilizing numerous data science tool-kits. In addition, (2) data science practitioners invest substantial effort in adapting implementations when switching between platforms or programming languages. Furthermore, (3) transitioning from "small" to "big" datasets often requires additional work, including the deployment of complex data structures, adoption of new libraries, and potential re-implementation.

This dissertation targets resolving these three issues while expediting scripting for developers. We utilized low-code techniques and Machine Learning (ML)-based approaches to accelerate programming tasks. Additionally, we deployed multiple libraries with domain-specific operations to simplify implementation tasks when transitioning across platforms. Moreover, within these libraries, we standardized Application Programming Interfaces for both sequential and parallel processing, enabling users to seamlessly switch between those two with ease. Accordingly, this doctoral project emphasizes both research contributions and practical applications.

In practical terms, we developed a Visual Studio Code extension called *NLDSL* to support the development and utilization of Domain Specific Languages (DSLs), particularly for data analysis and processing. This extension simplifies scripting tasks for end-users and developers by harnessing the benefits of natural language-like DSLs. Users can readily reuse customized DSLs through shared DSL templates. Specifically, these DSLs employ unified grammars for both sequential and parallel operations to address scalability concerns. The extension has received positive feedback from the community, underscoring the need for such extension types.

Our research contributions primarily focus on accelerating programming with ML code generation techniques and enriching the above libraries for reproduction via published source code and data. We conceived and evaluated an ensemble of code recommenders, named *Extended Network*, to illustrate the enhanced accuracy achieved by the ensemble-like architecture. Besides, we deployed a refined evaluation method, *CT3*, to reveal valuable insights while comparing code completion approaches, a task often hindered by classical aggregated evaluation. Finally, we proposed *One-shot Correction*, a procedure to integrate user feedback into generative Artificial Intelligence models without explicit re-training, facilitating in-depth analysis of unexpected outcomes. The effectiveness of these methods was demonstrated through our empirical studies.

Zusammenfassung

Im vergangenen Jahrzehnt hat die Disziplin Data Science bemerkenswerte Fortschritte gemacht, was sich in der Verbreitung datengetriebener Strategien sowie dem raschen Wachstum von Arbeitsplätzen und der Erweiterung universitärer Lehrpläne in diesem Bereich äußert. Folglich festigt sich die Position von Daten als eines der wichtigsten Güter von Organisationen. Allerdings gehen mit umfangreichem Nutzen auch erhebliche Herausforderungen einher.

Insbesondere (1) werden viele Fachkräfte, die in ihrem jeweiligen Bereich Experten sind, denen aber Programmierkenntnisse fehlen, mit Schwierigkeiten beim Erlernen und Nutzen zahlreicher Data Science Toolkits konfrontiert. Zusätzlich (2) investieren Data Science Anwender substanzielle Anstrengungen in die Adaption von Implementierungen beim Wechsel zwischen Plattformen oder Programmiersprachen. Weiterhin (3) erfordert der Übergang von „kleinen“ zu „großen“ Datensätzen oft zusätzliche Arbeit, inklusive des Einsatzes komplexer Datenstrukturen, der Nutzung neuer Bibliotheken und potentieller Neuimplementierung.

Diese Dissertation zielt darauf ab, diese drei Probleme zu lösen und gleichzeitig das Erstellen von Skripten für Entwickler zu erleichtern. Wir nutzen Low-Code-Techniken und auf Machine Learning (ML) basierende Ansätze, um Programmieraufgaben zu beschleunigen. Des Weiteren haben wir mehrere Bibliotheken mit domänenspezifischen Operationen entwickelt, um Implementierungsaufgaben beim Übergang zwischen Plattformen zu vereinfachen. Innerhalb dieser Bibliotheken haben wir darüber hinaus Programmierschnittstellen sowohl für sequentielle als auch parallele Prozessierung standardisiert, um Nutzern den nahtlosen und einfachen Wechsel zwischen beiden zu ermöglichen. Diese Doktorarbeit fokussiert sich auf Beiträge zur Forschung sowie praktische Anwendungen.

Im praktischen Bereich haben wir eine Visual Studio Code Erweiterung namens *NLDSL* bereitgestellt, um die Entwicklung und Nutzung von Domain Specific Languages (DSLs), insbesondere für die Datenanalyse und –verarbeitung, zu unterstützen. Diese Erweiterung vereinfacht Skripting-Aufgaben für Endanwender und Entwickler, indem sie die Vorteile von DSLs, die der natürlichen Sprache nahe sind, nutzt. Durch gemeinsam genutzte DSL-Vorlagen können Benutzer angepasste DSLs problemlos wiederverwenden. Insbesondere nutzen diese DSLs vereinheitlichte Grammatiken für sowohl sequentielle als auch parallele Operationen, um Skalierbarkeitsprobleme zu adressieren. Die Erweiterung hat in der Community positive Resonanz erhalten, was die Notwendigkeit solcher Erweiterungen betont.

Unsere Forschungsbeiträge fokussieren sich hauptsächlich auf die Beschleunigung der Programmierung mit ML-Techniken zur Code-Generierung und die Anreicherung der oben genannten Bibliotheken zur Reproduktion über veröffentlichte Quellcodes und Daten. Wir haben

ein Ensemble von Code-Empfehlern namens *Extended Network* konzipiert und evaluiert, um die Verbesserung der Genauigkeit durch die Ensemble-Architektur zu veranschaulichen. Außerdem haben wir eine verfeinerte Evaluierungsmethode, *CT3*, implementiert, um wertvolle Einblicke beim Vergleich von Ansätzen zur Code-Vervollständigung zu gewinnen, eine durch klassische aggregierte Evaluation oft beeinträchtigte Aufgabe. Abschließend haben wir *One-shot Correction* eingebracht, ein Verfahren zur Integration von Benutzer-Feedback in generative AI-Modelle ohne explizites Neu-Training, um eine eingehende Analyse unerwarteter Ergebnisse zu ermöglichen. Die Effektivität dieser Methoden wurde durch unsere empirischen Studien demonstriert.

Acknowledgements

Accomplishing a Doctoral degree rewards one not only with knowledge and precious experience in solving scientific problems, but also with effective methods in handling obstacles throughout the entire journey. This dissertation is a culmination of the efforts and support of many individuals, and I am sincerely thankful to each and every one of them.

First and foremost, I would like to express my deepest gratitude to my supervisor, **Prof. Dr. Artur Andrzejak**, for his enthusiasm, timely support, and endless patience through the whole itinerary. His essential guidance, immense knowledge, and plentiful experience have encouraged me at all time during my research. I have also acquired substantial insights from him, which helped me advance both professionally and personally.

My genuine appreciation extends to the German Academic Exchange Service (i.e. Deutscher Akademischer Austauschdienst, **DAAD**) for funding my research. Their comprehensive support allowed me to focus on my study without the burden of financial constraints. Furthermore, workshops and academic events organized by DAAD also offered me priceless opportunities to connect and exchange knowledge with other scholarship holders.

I am grateful to my colleagues at the Artificial Intelligence for Programming (**AIP**) group. Their contributions in developing ideas and providing technical support facilitated the project development. In particular, I thank **Kevin Kiefer** for deploying the core function of the *NLDSL* extension, developing the Cosy repository to explore Transformers with TensorFlow, and for enthusiastically explaining technical concepts to me. I thank **Patrick Weber**, **Philipp Walz**, **Jona Neef**, and **Dennis Pfleger** for contributing useful features to the *NLDSL* extension. I thank **Tim Waibel** for his responsibility and constructive discussions while designing enhanced features for the *NLDSL* extension. I thank **Christopher Höllriegl** for his expert teamwork while fixing bugs of the *NLDSL* extension with me. I extend my thanks to **Max Eric Henry Schumacher** and **Gabriel Rashidi** for their hard work and for intensive discussions while preparing our publications for the *Extended Network* and *CT3* approaches. I thank **Guayong Weng**, **Niklas Loeser**, and **Lennart Stöpler** for their delightful discussions during our weekly meetings in the last year of my PhD progress. I thank **Min Xue** for offering me a joyful company in the office, particularly after an extended period of working alone due to the Covid-19 situation.

My special thanks go to **Dr. Mohammadreza Ghanavati**, a former Doctoral student in our group, for promptly assisting me in upgrading the group's website, as well as for his valuable advice from pursuing a PhD to securing employment. I also would like to express my sincere thanks to **Dr. Anja Kleebaum**, graduated from the Software Engineering group, who worked near my office and generously shared her thesis with me upon knowing I was writing my

dissertation. The LaTeX configurations utilized in her thesis provided significant inspiration for formatting my own dissertation.

I am genuinely thankful for the modern amenities, kindly support, and useful courses organized by the **Heidelberg University**. With access to both hardware and software necessary for my research, I was well-equipped to pursue my studies. The courses conducted by the Graduate Academy of the university served as helpful supplementary sources for my individual Doctoral project. Additionally, their workshops focusing on soft skills have been highly beneficial in aiding me to effectively convey my methodologies, both verbally and textually. I am also grateful to **Ms. Stephanie Köhl** from the Doctoral Office for her enthusiastic support in managing my doctoral application and for her practical information on preparing my dissertation.

Ultimately, I express my heartfelt gratitude to my **families** and **friends** for their unconditional love, never-ending support, and persistent encouragement. Their reassurance strengthened my determination in pursuing this academic degree. Specifically, my earnest thanks to my partner, **Mr. Lars Ritter**, for his tremendous understanding, tireless proofreading, and incalculable effort in comforting me with his flavorful cuisine whenever I encounter difficulties. My sincerest thanks to **Mr. Ritter's family** for their incredibly warm welcome, genuine care, and heartfelt support, which have made me feel like a cherished member of the family, significantly easing my life in Germany. Finally, my deepest appreciation to **my parents** for their wholehearted care, constant encouragement, and unwavering support throughout my journey, despite being almost ten thousand kilometers away from them. They are the best parents anyone could ask for.

Antoine de Saint Exupéry wrote:

> *Grown-ups love numbers.*
>
> *When you tell them that you have made a new friend, they never ask you any questions about essential matters. They never say to you "What does his voice sound like? What are his favorite games? Does he collect butterflies?". Instead, they ask you: "How old is he? How many brothers does he have? How much does he weigh? How much money does his father make?". Only from these numbers do they think they know anything about him.*
>
> *If I say to the grown-ups, "I saw a beautiful house made of pink-coloured bricks, with geraniums in the windows and doves on the roof", they will not be able to picture the house at all. You have to say, "I saw a house worth 100,000 francs". Then they'll exclaim, "Oh, how nice!".*
>
> —The Little Prince, Saint-Exupéry, 2016

These words constantly crossed my mind as I conducted my research.

Contents

Abstract v

Zusammenfassung vii

Acknowledgements ix

List of Figures xix

List of Tables xxiii

List of Listings xxv

I Preliminaries 1

1. Introduction 3

1.1. Motivation . . . 3
1.2. Research Goals . . . 4
1.3. Research Strategy . . . 5
1.4. Core Research Questions . . . 7
1.5. Contributions . . . 8
1.6. Publications . . . 10
1.7. Structure of the Dissertation . . . 12

2. Background and Related Work 15

2.1. Facilitating Data Analysis and Processing in Low-code Manner . . . 15
2.1.1. Low-code and No-code Development . . . 16
2.1.2. Domain-Specific Languages . . . 19
2.1.3. Low-code Tool-kits for Data Analysis and Processing . . . 24
2.2. Advancing Programming with Machine Learning-based Approaches . . . 27
2.2.1. Application Fields . . . 27
2.2.2. The Naturalness Hypothesis . . . 30
2.2.3. Traditional Machine Learning Methods for Code Generation . . . 31
2.2.4. Deep Learning Techniques for Source Code . . . 35

2.3. Transformers and Beyond . . . 42
2.3.1. Vanilla Transformer . . . 42
2.3.2. Successors of the Transformer Model . . . 49
2.4. Summary . . . 51

II Practical Contributions **53**

3. NLDSL Extension:
Accelerating Programming for Data Analysis Tasks with Low-code Approaches in Practice **55**
3.1. Introduction . . . 55
3.2. Python-based (vanilla) NLDSL Tool . . . 57
3.2.1. Tool Architecture . . . 58
3.2.2. DSL Structure . . . 58
3.2.3. Core Functionalities . . . 59
3.3. NLDSL Visual Studio Code Extension . . . 64
3.3.1. DSL Development . . . 65
3.3.2. Code Completion-related Features . . . 68
3.3.3. Utilities . . . 72
3.4. Dissemination . . . 73
3.4.1. Managing CI and CD Pipelines on Azure . . . 74
3.4.2. Build Instructions in YAML Files . . . 75
3.5. Discussion . . . 76
3.5.1. Preliminary Evaluation . . . 76
3.5.2. Potential Enhancements . . . 80
3.5.3. Response to CRQ1 . . . 81
3.6. Summary . . . 82

III Research Contributions **85**

4. Extended Network:
Improving Code Recommendations by Combining Neural and Classical ML Approaches **87**
4.1. Introduction . . . 87
4.2. Background and Related Work . . . 89
4.2.1. The Naturalness Hypothesis . . . 89
4.2.2. Typical Machine Learning Models for Code Completion . . . 89
4.2.3. PHOG and Pointer Mixture Network . . . 90
4.3. Extended Network . . . 91
4.3.1. The Core Idea . . . 91
4.3.2. An Illustrative Model for the Extended Network Architecture . . . 93
4.3.3. Component Selection in the Extended Network Model . . . 94

4.4. Evaluation . . . 96
4.4.1. Experimental Setup . . . 96
4.4.2. Experimental Results . . . 99
4.4.3. Response to CRQ2 . . . 103
4.5. Summary . . . 103

5. Code Token Type Taxonomy:
A Methodology for Refined Evaluation of ML-based Code Completion Approaches **105**
5.1. Introduction . . . 105
5.2. Background and Related Work . . . 109
5.2.1. Machine Learning for Code Completion . . . 111
5.2.2. Aggregated and Refined Metrics for Evaluation . . . 112
5.2.3. Out-of-Vocabulary Issue . . . 115
5.3. Code Token Type Taxonomy . . . 115
5.3.1. General Workflow . . . 115
5.3.2. CT3 Schema for Python . . . 118
5.3.3. Open Vocabulary for Transformers . . . 123
5.4. Evaluation . . . 124
5.4.1. Research Questions . . . 124
5.4.2. Experimental Setup . . . 124
5.4.3. Evaluation Results . . . 127
5.5. Discussion . . . 132
5.5.1. CT3 Challenges . . . 133
5.5.2. Threats to Validity . . . 133
5.5.3. Response to CRQ3 . . . 134
5.6. Auxiliary Experiments . . . 134
5.6.1. Length Distribution of Terminal Tokens in Python150k . . . 135
5.6.2. Length Threshold for Open Vocabulary Building . . . 135
5.6.3. Length Threshold for Input Data File Creation in Open Vocabulary Case 137
5.6.4. Window Size for Input Data File Creation in Open Vocabulary Case . . . 139
5.7. Summary . . . 140

6. One-shot Correction:
Enhancing Code Generation Models through User Feedback and Decomposition Techniques **141**
6.1. Introduction . . . 142
6.2. Background and Related Work . . . 144
6.2.1. Generative Artificial Intelligence for Code . . . 144
6.2.2. Interactive Programming . . . 145
6.2.3. Decomposition in Problem Solving . . . 145
6.2.4. Chunking in Natural Language Processing . . . 146
6.3. One-shot Correction . . . 146
6.3.1. General Workflow . . . 146

6.3.2. Query Chunking . . . 148
6.3.3. Sub-snippets Retrieving/Generating . . . 149
6.3.4. Code Building . . . 151
6.4. Experiments . . . 155
6.4.1. Research Questions . . . 155
6.4.2. Experimental Setup . . . 156
6.4.3. Evaluation Metrics . . . 160
6.5. Evaluation Results . . . 160
6.5.1. Evaluation Results by Difficulty Level . . . 160
6.5.2. Ablation Study . . . 165
6.5.3. LLM Involvement . . . 168
6.6. Discussion . . . 171
6.6.1. Threats to Validity . . . 171
6.6.2. Challenges and Potential Enhancements . . . 172
6.6.3. Response to CRQ4 . . . 173
6.7. One-shot Correction GUI . . . 174
6.7.1. An Overview of the One-shot Correction GUI . . . 174
6.7.2. A Demo of Main Features . . . 175
6.8. Summary . . . 180

IV Conclusions **181**

7. Summary **183**

8. Future Work **187**
8.1. Improvements for Proposed Approaches . . . 187
8.2. Potential Future Research Directions . . . 188
8.2.1. Code to Code Translation . . . 188
8.2.2. Knowledge-enhanced Large Language Models . . . 188
8.2.3. Addressing Transformers' Shortcomings . . . 189

V References **191**

List of Acronyms **193**

Bibliography **195**

VI Appendix **213**

A. Data Science-related Jobs:

A Glimpse of The Past Decade **A1**

A.1. The Rise of Data Science-related Jobs A1

A.2. Time Allocation for a Data Scientist A2

A.3. Technical and Analytical Know-how Problem A3

B. Artificial Neural Network:

Fundamental Concepts and Techniques **B1**

B.1. Overview of Artificial Neural Networks B1

B.2. Feed-forward and Recurrent Neural Network B2

B.3. Problems with Recurrent Neural Networks B3

B.4. Long Short-Term Memory B4

B.5. Word Embedding B5

B.6. Variants of the Attention Mechanism B6

List of Figures

1.1. Interactive construction of programs/workflows with the accelerated programming (AP) tool targeted in this dissertation. 5

2.1. UML class diagram* of Java code used to create an *internal* DSL (Sanaulla, 2013). 21
2.2. Underlying concept of *low-code* approaches for data analysis and processing, adapted from Figure 1 of Makonin et al. (2016). 25
2.3. A simple example of generating code token sequences with *n-gram* models. 30
2.4. A CFG for binary expressions with four operations (a), alongside a parse tree (b), and an AST (c) of an example expression (Stanford, 2015; Pingali, 2023). 32
2.5. An example illustrating the difference in employing PHOG and PCFG for code completion, adapted from Figure 1 of Bielik et al. (2016). 34
2.6. Abstract representation of the encoder-decoder architecture through time steps with an illustrative example. 36
2.7. Abstract representation of the attention mechanism decoding at time step t, adapted from Figure 1 of Bahdanau et al. (2015) and Figure 4 of Weng (2018). . 37
2.8. An example of using Pointer Network in plantar convex hull problem, adapted from Figure 1(b) of Vinyals et al. (2015). 39
2.9. Predicting a next code token with Pointer Mixture Network at time step t, adapted from Figure 3 of Li et al. (2018). 40
2.10. Transformer at an abstract-level view when generating an output sequence (a) and when being trained for predicting target output word $tout_i$ (b), adapted from Alammar (2018a) and Figure 1 of Vaswani et al. (2017). 42
2.11. A deeper look at encoder and decoder layers of Transformer, adapted from Figure 1 of Vaswani et al. (2017). 43
2.12. Architecture inside the multi-head attention of Transformer, adapted from Figure 2 of Vaswani et al. (2017). 45
2.13. Utilizations of multi-head attention in Transformer architecture. 46

3.1. Architecture of *NLDSL* tool, adapted from Figure 1 of Andrzejak et al. (2019a). . 58
3.2. Grammar of *evaluation* DSL, adapted from Figure 2 of Andrzejak et al. (2019a). 58
3.3. Code completion at operation level with *NLDSL* takes place at the beginning of the pipeline (a), and after an initialization operation (b). 60
3.4. Code completion at statement level with *NLDSL*. 60

3.5. Code completion at operation level after defining a new DSL function. 64
3.6. An overview of advanced features delivered with *NLDSL* extension. 64
3.7. An example of Excel template for DSL creation with *NLDSL*. 66
3.8. *NLDSL* menu in VSCode after adding a new DSL named `example_dsl`. 66
3.9. Column names completion with type provider in *NLDSL**. 69
3.10. Path completion supported by *NLDSL*. 70
3.11. Setting *target code* and initializing libraries with *NLDSL*. 70
3.12. Recalling DSL grammar for `apply` operation while editing with *NLDSL*. 71
3.13. General steps for disseminating the *NLDSL* extension with Azure Pipelines. . . . 74
3.14. Workflow of activating CI and CD pipelines while releasing new extension features. 74
3.15. Tasks specified in YAML files for CI pipelines. 75
3.16. Statistics on installation numbers of three variants of *NLDSL* extension over time. 78

4.1. Flowchart displaying the component selection in the *Extended Network* during prediction, adapted from Figure 3.4 of Schumacher (2019). 92
4.2. An example of generating a label for terminal `T` and writing it to the terminal corpus `tcorpus`, adapted from Figure 3.6 of Schumacher (2019). 94

5.1. An example for evaluating a code snippet using aggregated accuracy. 106
5.2. An example for evaluating a code snippet using refined accuracy. 107
5.3. Implementation and usage of *Code Token Type Taxonomy (CT3)*. 116
5.4. Frequency intervals for determining frequency labels of code tokens with *CT3*, adapted from Figure 4.3 of Rashidi (2021). 123
5.5. Comparing the accuracy of the closed and open vocabulary models for the syntax type dimension (a) and its token types distribution (b). 129
5.6. Comparing the accuracy of the closed and open vocabulary models for the origin dimension (a) and its token types distribution (b). 130
5.7. Comparing the accuracy of the closed and open vocabulary models for the length dimension (a) and its token types distribution (b). 130
5.8. Comparing the accuracy of the closed and open vocabulary models for the frequency dimension (a) and its token types distribution (b). 131
5.9. Top-15 common terminal token lengths in the Python150k dataset for the training 100k dataset (a) and the evaluation 50k dataset (b). 135
5.10. Top-15 common token lengths in the Python150k dataset (a) – (b) and the JavaScript150k dataset (c) – (d). 138

6.1. A simple scenario of interactive programming, highlighting the issue of utilizing user feedback across sessions. 143
6.2. *One-shot correction* workflow for NL-to-Code translation models, exemplified with an illustrative example. 147
6.3. A dependency graph generated by spaCy. 148
6.4. Flowchart of retrieving/generating sub-snippets for an NL chunk. 149

6.5. An example of building code from sub-snippets* . . . 151
6.6. CodeBLEU scores by difficulty level on all test cases (left), on correct test cases (middle) and on incorrect test cases (right) of the chunking methodology. . . . 161
6.7. CodeBLEU scores by complexity level for our CodeGenC model. . . . 166
6.8. Correct outcome ratio for specific cases. . . . 168
6.9. Correct outcome ratio for certain cases with GPT35Prompt taken into account. . . 170
6.10. General scenario of using the *One-shot Correction* GUI. . . . 174
6.11. Searching the example input query for the first time. . . . 176
6.12. Recommended steps to customize the displayed code snippet. . . . 177
6.13. Saving the correction and inquiring the same query again. . . . 178
6.14. Activating the highlight matching feature. . . . 178
6.15. Generating code snippet for the same input query after manipulating queries in the correction data-store. . . . 179

A.1. Interest over time* of data science-related jobs from 2013 to 2023 via Google Trends. A2
A.2. Summary of surveys from multiple sources from 2016 to 2022** on the question "How do data scientists spend their time?". . . . A3

B.1. An example of ANN with two hidden layers. . . . B1
B.2. The flow of information in an RNN handling a sequence of inputs with length `seqlen`, adapted from Figure 5.2 of Kelleher (2019). . . . B3
B.3. Unrolling the RNN architecture depicted in Figure B.2 through time, adapted from Figure 5.3 of Kelleher (2019). . . . B3
B.4. The flow of information within an LSTM cell at time step t, adapted from Figure 5.4 of Kelleher (2019). The input and output layers are omitted. . . . B4

List of Tables

1.1. A mapping from chapters to target problems and core research questions (CRQs). 12
1.2. A summary of predefined icons used in the dissertation. 14

2.1. A survey of how data scientists spend their time (Anaconda, 2022). 22
2.2. The formal definition of Context-Free Grammar (Jurafsky et al., 2008). 31

3.1. Contributors in developing and disseminating the *NLDSL* extension. 83

4.1. Encoding labels in the terminal corpus of the *Extended Network model*. 97
4.2. Experiment configuration for the *Extended Network model*. 98
4.3. Accuracy on the test-set of a single layer *Extended Network model*, without dropout, trained for 7 epochs on train-test split, vocabulary size of $1k$. 99
4.4. Accuracies for a single layer *Extended Network model* and a two-layer model across different values of dropout, vocabulary size of $1k$. 100
4.5. Accuracy of the two-layer *Extended Network model* (20% dropout), compared to state-of-the-art probabilistic language models and the original Pointer Mixture Network. 100
4.6. Accuracies of components in the *Extended Network model**, including predictions for nodes without values (i.e. labeled as `EmptY`). 101
4.7. Accuracies of components in the *Extended Network model**, excluding predictions for nodes without values (i.e. labeled as `EmptY`). 102

5.1. State-of-the-art of code completion models, primarily from 2018 to 2021. 109
5.2. Criteria for a refined evaluation. 113
5.3. *CT3* and related works in supporting refined evaluations. 113
5.4. *CT3* schema proposed for Python. 116
5.5. An example of *CT3* data for code tokens in the Python150k dataset. 117
5.6. An example of a combination log of *CT3*-data and prediction results for code tokens in the Python50k dataset. 118
5.7. Explanation and examples for Syntax Type dimension of *CT3*. 120
5.8. Explanation and examples for Context dimension of *CT3*. 122
5.9. Experiment configuration for *CT3*. 127
5.10. Aggregated accuracy of closed and open vocabulary models. 127
5.11. Exploration of length thresholds for Python100k dataset's open vocabulary building. 136

5.12. Investigation of length thresholds for Python150k dataset's input data file creation. 137
5.13. Examination of window sizes for Python150k dataset's input data file creation. . 140

6.1. Examples of refining sub-snippets for a non-last NL chunk with *2*-NNs. 152
6.2. Experiment configuration for *One-shot Correction*. 159
6.3. Definitions for difficulty levels (diff.). 161
6.4. CodeBLEU by difficulty level (diff.) across all approaches. 162
6.5. Examples of CodeGenE overlooks or gets confused by extra information. 163
6.6. Definitions for complexity levels (comp.). 166
6.7. CodeBLEU by complexity level across all test cases for CodeGenE model. 167
6.8. Correct outcome ratio and CodeBLEU for each model. 167
6.9. Correct outcome ratio and CodeBLEU for each model over test cases with multi-chunk queries and non-empty correction data-store. 170
6.10. *1*-NN of the input query at the beginning. 176
6.11. *1*-NN of the input query after updating the correction data-store. 179

B.1. Meaning of data parameters in Figure B.4. . . . . B5
B.2. Variants of attention architectures in Bahdanau et al. (2015) and Luong et al. (2015). B6

List of Listings

2.1. Creating a graph with the *internal* DSL, i.e. methods from classes `GraphBuilder` and `EdgeBuilder`. . . . 21
2.2. Creating a graph without the *internal* DSL, i.e. excluding classes `GraphBuilder` and `EdgeBuilder`. . . . 21
2.3. SQL query. . . . 22
2.4. *NLDSL* statement. . . . 22
2.5. Translating NL to Python code with GitHub Copilot (v1.155.0) in VSCode. . . . 28
2.6. Generating explanation for the code snippet in Listing 2.5 (lines 3 – 6) with GitHub Copilot (v1.155.0) and Copilot Chat (v0.11.1) in VSCode. Just a part of the text is presented for demonstration purpose. . . . 29
2.7. Completing Python code with GitHub Copilot (v1.155.0) in VSCode. . . . 29

3.1. An example of a DSL statement for Pandas. . . . 56
3.2. Grammar for the left-hand side of *definition* DSL statements in EBNF, adapted from Andrzejak et al. (2019a). . . . 59
3.3. Specifying *target code* as `pandas` for translating subsequent DSL statements into Pandas code. . . . 61
3.4. Dataframe inspection with Pandas DSL. . . . 61
3.5. Data preprocessing with Pandas DSL. . . . 62
3.6. Documentation defined for DSL operation `group by`, adapted from the source code of vanilla *NLDSL* (Andrzejak et al., 2019a). . . . 62
3.7. Expression rule for DSL operation `group by`, adapted from the source code of vanilla *NLDSL* (Andrzejak et al., 2019a). . . . 63
3.8. Defining a new DSL operation via internal function. . . . 63
3.9. Definition for the DSL operation `group by` in `tx` format. . . . 65
3.10. Basic Deep Learning tasks with TensorFlow DSL*. . . . 67
3.11. Documentation defined for DSL operations `group by` and `apply` in *NLDSL* extension v0.5.0. . . . 68
3.12. Usages of `group by` and `apply` operations with adjusted grammars. . . . 68
3.13. Specifying a CSV file for type provider feature. . . . 69
3.14. Initializing relevant libraries for `spark` *target code* in *NLDSL* extension. . . . 70
3.15. Configuration of auto-imported libraries for each *target code*. . . . 71
3.16. Handlers registered for `did_change` event in *NLDSL*. . . . 72

3.17. Synchronization indicator with *NLDSL*. 73
3.18. Setting up a standalone Python interpreter for the Linux-*NLDSL* extension. . . . 76

5.1. An example code snippet for extracting syntax type information. 119
5.2. An example code snippet for extracting context information. 121

6.1. Message for translating NL query/chunk to Python code using GPT-3.5-Turbo-0301. 157
6.2. Message for translating NL query/chunk to Python code using GPT-3.5-Turbo-0301 with correction information combined in input queries. 158
6.3. Prompt template for translating NL query to Python code using GPT-3.5-Turbo-0125 and our chunking strategy as task descriptions. 169
6.4. Relevant fields of the configuration file on handling the state of the correction data-store and refining code token types. 175

Part I

Preliminaries

Introduction

Chapter 1

This dissertation aims to address the challenge of supporting researchers and practitioners in effective programming of data analysis and processing tasks. The first chapter introduces our motivation (Section 1.1), research goals (Section 1.2), and the corresponding research strategy (Section 1.3). Subsequently, core research questions derived from the strategy are described in Section 1.4. Contributions of this thesis and our publications are presented in Sections 1.5 and 1.6, respectively. Finally, the structure for the rest of the manuscript is outlined in Section 1.7.

1.1 Motivation

In the past decade (2013 – 2023), data has become one of the most valuable properties of any organizations due to the speedy development of technologies and the exponential growth of data volume (Provost et al., 2013; Alharthi et al., 2017). A statistic published by Taylor (2023) reveals that the amount of created, consumed, and stored data worldwide is expected to reach 120 Zettabytes[1] by 2023, exceeding the volume in 2013 by a factor of 13.3.

This data explosion leads to an urgent demand of agile and effective data analysis to extract meaningful insights for diverse enterprises, which is also known as *data-driven decision making* (Anderson, 2015; Tabesh et al., 2019; Mikalef et al., 2021; Acciarini et al., 2023). Consequently, *data science*, a combination of multiple disciplines including math and statistics, specialized programming, advanced analytics, Artifical Intelligence (AI), and Machine Learning (ML)[2], has captured intensifying attention in recent years.

However, the accelerated progress of data science has unveiled not only a myriad of chances but also unneglectable challenges. Notably, most end-users struggle with learning and using a diversity of data science tool-kits due to the technical and analytical know-how problem (BARC, 2015; Knime, 2023). As a result, facilitating data analysis and processing for domain experts, who are proficient in specific domains (e.g. natural science, engineering, etc.) but have little programming experience, is a significant challenge in data science.

Andrzejak et al. (2019b) pointed out three common problems encountered by data scientists and practitioners at the intersection of data science and software engineering. These issues are (i) *programming barrier* </>, (ii) *reuse problem*, and (iii) *scalability problem*, subsequently elaborated as follows:

Programming barrier </>. Typically, data scientists must engage a substantial amount of programming languages (e.g. Python, R, SQL), frameworks (e.g. Pandas, NumPy, Scikit-learn),

[1] 1 Zettabyte is 10^{12} Gigabytes or 2^{70} Bytes.
[2] What is data science, https://www.ibm.com/topics/data-science, (Accessed: 02 November 2023).

and tools (e.g. Tableau[3] and SAS[4]) for various data types such as text, time-series data, and still images (CrowdFlower, 2015; FigureEight, 2018; Appen, 2019; Anaconda, 2022). A project-specific analysis pipeline can involve numerous programming languages, frameworks, and tools, which require a certain knowledge from end-users. Furthermore, each language or tool-kit usually comes with its own Application Programming Interfaces (APIs) and implicates a considerable effort to find and to understand the suitable functions or components from available libraries. All these combined significantly prolong a project's duration and inflate its budget, while hindering data scientists and practitioners in processing data faster and more efficient.

Reuse problem . With the wide adoption of ML in data science, a majority of the effort from data scientist and practitioners is dedicated to adjusting code of well-known algorithms and techniques to specific requirements of a project and its datasets. This necessitates high-quality reusable libraries. Additionally, deploying ML models across different programming languages also broadens this problem. Recent surveys published by Anaconda (2020; 2021; 2022) reveal the top five roadblocks of respondents when moving their models to a production environment, including the barriers of re-coding ML models from Python/R to another language, and vice versa. Reusable libraries that can efficiently switch between languages would be a promising direction to tackle this issue.

Scalability problem . In 2018, 52.18% respondents of a survey conducted by Anaconda (2018) confirmed that they did not use any technologies for scaling out their data science. This is no longer feasible with the explosion of data. Ensuring scalability is now critical and considered as one of the obstacles preventing organizations from successfully integrating ML models into their software development life-cycle (Knime, 2023). However, scaling algorithms and software pipelines for large datasets and millions of users in a cost-effective and time-saving way can be challenging. It involves more complex data structures, libraries, and in some cases even re-implementation (Andrzejak et al., 2019b). Therefore, a seamless transformation between sequential and parallel processing scripts for data scientists and practitioners would be beneficial under these circumstances.

In this dissertation, we aim to mitigate the three aforementioned problems by proposing a set of approaches and prototypical tools. These solutions target to enable the creation of software pipelines for scalable data analysis and processing, in an interactive, user-friendly way.

1.2 Research Goals

The purpose of our research project is to effectively and efficiently address the *programming barrier* , *reuse problem* and *scalability problem* described above. A conceptual goal to tackle the *programming barrier* is to develop effective methods for intelligent code recommendations for scripting data analysis workflows. The *scalability problem* is mitigated by developing a common API and (for selected cases) Domain-Specific Languages (DSLs). In this way, users can write the same code for "small data" and "big data" scenarios. The project

[3] Business intelligence and analytics software, https://www.tableau.com/.
[4] Statistical Analysis Software, https://www.sas.com/en_us/home.html. (Accessed: 22 November 2023).

focuses on research contributions but also devotes itself to the practical side through prototypical tools and some libraries. The latter provide partial solutions to the *reuse problem* .

Particularly, this dissertation attempts to deliver three following complementary components for scalable data integration, processing and analysis with consideration of software reuse:

(A) A set of methodologies for accelerated development of flexible software pipelines. This will address assistance for users with little programming experience but also facilitate development of pipelines by experts.

(B) A collection of libraries with functions/algorithms for multiple domains of data analysis and processing, as well as tools for creating further analogue libraries.

(C) Support for scalable processing of data via an approach to access sequential or parallel version of algorithms under a uniform API and via massively-parallel implementations of suitable functions from (B).

The next subsection discusses these components in detail alongside our strategy to alleviate the three target problems.

1.3 Research Strategy

A possible application for the components mentioned in Section 1.2 involves interactive creation of programs or workflows, as depicted in Figure 1.1. While users explore data, ❶ an accelerated programming (AP) tool suggests them most likely procedures for data transformations or aggregations. The latter are selected by ML methods (*component A*) based on previous data explorations, encoded knowledge of domain experts[5], and a set of user-defined functions extracted from the *component B*. Furthermore, the sequence of operations is presented in form of an easily interpretable language, such as Domain-Specific Language (DSL) or Natural Language (NL).

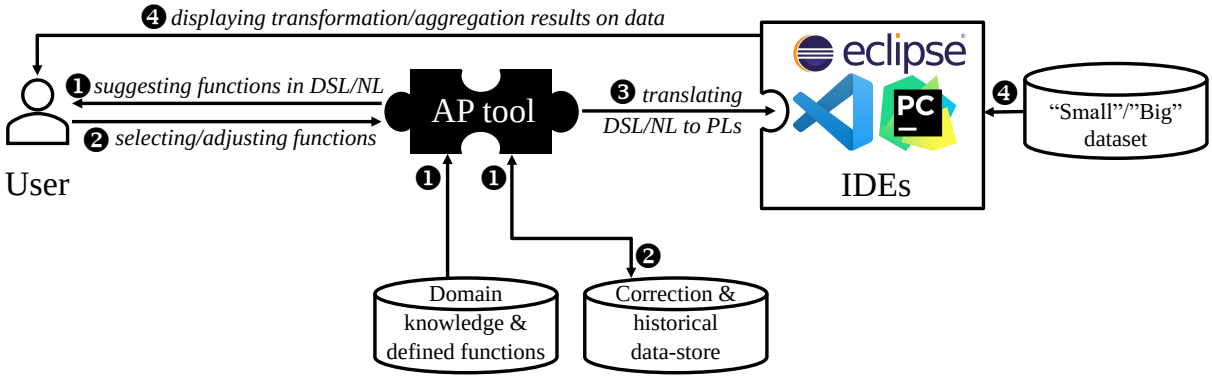


Figure 1.1: Interactive construction of programs/workflows with the accelerated programming (AP) tool targeted in this dissertation.

During data exploration, ❷ users select and possibly adjust a suitable operation, which will be recorded in a data-store for future reference. Subsequently, ❸ the chosen operation is

[5]Users who excel in specific domains but have little programming experience.

translated to a General-purpose Programming Language (GPL) such as Python or R, and ❹ the results of this operation are readily performed on a dataset via Integrated Development Environments (IDEs). This gives immediate feedback in terms of correctness of the suggested functions and creates hints for further steps. In other words, the AP tool should be fully integrated into IDEs. Ultimately, another essential feature of such AP tools is that they can be executed in a distributed way (e.g. with Apache Spark[6] as back-ends) to handle very large datasets while require minimum effort from users (*component C*).

We anticipate that the above concept can be particularly effective in reducing the programming effort for data scientists and practitioners in their big data analysis tasks, specifically as follows:

> **Mitigating programming barrier** </> by harnessing (i) advanced code recommendation methods, (ii) "high-level programming", and (iii) a data-centric development environment.

Code recommendation methods support users in selecting the next step of processing by proposing most likely code fragments or macros given one or more "hints". ML and AI techniques are exploited to suggest more complex programs. The overall effect is that less experienced users do not need to know/learn available processing operations and can "try out" until the desired result is achieved.

Besides, the "high-level programming" is enabled via libraries of domain-specific operations, represented as embedded DSLs, which integrate seamlessly into the underlying programming language (e.g. Python). Each library will capture essential high-level operations in a respective domain (e.g. data exploration with Pandas/Spark, model creation with TensorFlow/PyTorch).

Finally, a data-centric development environment assists automatic preview of the effects of each coding step on data. This approach can aid users with program comprehension (understanding what the code is performing on the data) and program verification (checking whether each operation yields expected results). In our project, we leverage modern IDEs for this feature.

> **Alleviating reuse problem** ⟳ by creating multiple libraries of domain-specific operations and macros to facilitate accelerated programming of software pipelines in these domains.

The domains covered in this dissertation include: (i) extraction and integration of data from heterogeneous databases, (ii) building, training, and evaluating standard ML models, and (iii) evaluating ML models in a refined manner to gain insights for improvements. Libraries of the two former domains can also be created and extended through a supplementary wizard to facilitate the progress. This wizard supports users in forming their own domain-specific operations based on predefined samples.

> **Tackling scalability problem** ⇲ by (i) unifying APIs for sequential and massively-parallel processing, and (ii) implementing selected approaches from the aforementioned domains of *reuse problem* ⟳ in a scalable fashion.

The first target is achieved by enhancing the base language (Python or R) and libraries with

[6] Unified engine for large-scale data analytics, https://spark.apache.org/, (Accessed: 04 December 2023).

an embedded DSL for processing data frames (i.e. table-like data structures) or relational tables. There is a dual implementation for these operations, namely sequential and massively-parallel (under Apache Spark). Users are then able to switch between both modes without changing the DSL commands of these operations.

The second target is accomplished by implementing selected suitable algorithms and functions from the last domain considered in the *reuse problem* as massively-parallel versions. These implementations should provide identical or similar APIs as for sequential versions. This will ensure scalability of specialized algorithms, which might be difficult to implement or inefficient in the proposed embedded DSL.

Methodologies and techniques proposed for addressing the *programming barrier* </>, *reuse problem*, and *scalability problem* are integrated into approaches detailed in Chapters 3 – 6. Each chapter focuses on specific parts of the problems, as summarized in Section 1.7. The next section presents our core research questions, which are extracted from the discussed strategy.

1.4 Core Research Questions

Given the considered research goals and strategy, our work is conducted by defining and exploring the following core research questions (CRQs):

> **CRQ1. Do embedded external Domain-Specific Languages (DSLs) offer benefits for implementing data analysis tasks?**

DSLs, the foundation of *low-code* development, are specialized languages that yield substantial gains in conveying business logic and ease of use compared to GPLs in their domain of application (Mernik et al., 2005). To answer this question, we first identify the obstacles which practitioners face while fulfilling their data analysis tasks, and then validate if these barriers can be mitigated using DSLs. Particularly, we divided this research question into sub-problems as follows:

- Determining the hurdles in data analysis tasks,
- Investigating DSLs and analyzing their advantages and challenges in deployment,
- Employing DSLs to accelerate programming in data analysis tasks with *low-code* manner.

The first two points are clarified in Chapter 2 with supplementary information from Appendix A. Meanwhile, Chapter 3 unveils solutions for the last point. We published a Visual Studio Code extension based on our proposed external DSL that has a custom syntax and is embedded into source code of GPLs as comment statements – *hence the term "embedded external DSLs"*[7].

The response to this question serves as our contribution to the practical side of the doctoral project. The remaining CRQs are dedicated to the research aspect.

> **CRQ2. Do ensembles of Machine Learning (ML)-based recommenders improve the accuracy for code completion approaches?**

[7] See Section 2.1.2 for the definition and examples of *external* DSLs.

Ensemble methods are considered as the state-of-the-art solution for many ML challenges, since they improve predictive performance of a single model by training multiple models and combining their predictions (Sagi et al., 2018). We address this question by leveraging this concept and evaluating an ensemble-like ML-based model for code completion in comparison to its components, i.e. single models of classical and neural ML approaches. Description of the model and our evaluation results are presented in Chapter 4.

> **CRQ3. Do traditional aggregated evaluation methods reveal useful information for comparing and characterizing code completion approaches?**

Given that ensemble-like methods can enhance the prediction accuracy of single models, it is important to analyze the effect of each component within the ensemble on specific cases, and to thereby find hints for further improvements. However, this information is usually omitted in the results obtained by traditional aggregated evaluation methods, which combine results over all types of code tokens. We tackle this question by proposing a methodology for refined evaluation, breaking down the performance of a model into each category of code token types. Chapter 5 demonstrates the methodology and its advantage in comparing and characterizing code completion approaches.

> **CRQ4. Can user feedback enhance Natural Language (NL) to code models without explicit re-training**?

NL to code translation is a prominent approach in facilitating programming. Despite this method gathering significant attention in recent years, users still encounter some specific problems in attaining their desired target code. Additionally, feedback from users is considered as a valuable resource but has been overlooked by current Artifical Intelligence (AI) models.

We answer this question by introducing a methodology for integrating user feedback into NL to code models without further re-training. The latter is ensured by incorporating a correction data-store into the code generation process. The utility of our method is illustrated by comparing the code generated solely by an AI code generator model to the code produced when embedding user feedback into the AI model. Chapter 6 discusses the concept and evaluation in detail.

1.5 Contributions

This dissertation makes the following contributions:

> **An extension facilitating the usage and development of embedded external Domain-Specific Languages (DSLs)**

We developed a Visual Studio Code extension named *NLDSL*[8] to illustrate the practical application of our research strategy (Section 1.3). The extension supports data analysis in Python with customized DSLs for common operations. The DSL lines are integrated directly

[8] Visual Studio Marketplace for *NLDSL* extension on three different operating systems, https://marketplace.visualstudio.com/publishers/PVS-IfI-Heidelberg-University-Germany, (Accessed: 07 December 2023).

into Python code as comments and then translated to Python during editing. Consequently, users can work with diverse data analysis libraries with a small set of DSL commands.

In addition, the extension is implemented with Language Server Protocol (LSP)[9] to reduce the effort in expanding to other programming languages (e.g. R) and editors (e.g. JupyterLab, PyCharm). The current version of the extension (i.e. *NLDSL* v0.5.0) covers essential functions for Pandas data analysis library and Apache Spark framework for large-scale data processing. TensorFlow and PyTorch support is included, but still in an alpha state. This contribution is also the answer for our first core research question (CRQ1). Details about the extension are presented in Chapter 3.

Extensive evaluation and analyses illustrating the leverage of ensemble-like methods in code completion for dynamically typed languages

To substantiate the advantage of ensemble-like methods in augmenting ML code completion models (CRQ2), we conducted comprehensive evaluation and analyses of the *Extended Network model*, a Python code completion model. Max Eric Henry Schumacher, a Bachelor student in our research group, proposed this model as a combination of neural and classical ML approaches, namely Pointer-Mixture Network (Li et al., 2018) and Probabilistic Higher Order Grammar (Bielik et al., 2016).

Particularly, we (i) assessed the accuracy of the model with different settings of relevant hyper-parameters and (ii) investigated the performance of each component under multiple dimensions to understand its respective strengths. The *Extended Network model* achieves improvements in accuracy over each of its stand-alone models, which demonstrates the potential of ensemble-like methods for code completion and recommendation in dynamically typed languages. Chapter 4 introduces the model, our evaluation setup and analysis results in detail.

A methodology for refined evaluation of ML-based code completion approaches

We proposed a methodology called *Code Token Type Taxonomy (CT3)* which breaks down the accuracy of a code completion model into various dimensions of code token types. Each dimension is a relevant attribute of code token that plays a significant role in code completion demand (e.g. syntax type, origin, or frequency). Every token type in a dimension is identified by analyzing the Abstract Syntax Tree (AST) of the code snippet.

CT3 offers additional analysis beyond traditional evaluation results, specifically highlighting the impact of each component in a completion model. This information subsequently facilitates insights into model challenges. To demonstrate the utility of our approach, we conducted an empirical study on a Transformer-based code completion approach with two variants, closed versus open vocabulary. The refined evaluation reveals intriguing results with better accuracy of the latter variant on multiple token types.

Furthermore, to consolidate the novelty of our methodology and to give an overview of the research progress in code completion from 2018 to 2021, we review the state-of-the-art of ML-

[9] Language Server Protocol, https://microsoft.github.io/language-server-protocol/, (Accessed: 07 December 2023).

based approaches in this field. The summary emphasizes the challenge of comparing between models without explicitly re-evaluating them, due to the differences on numerous aspects such as input representations, used datasets, and evaluation metrics. This leads to a demand for a set of standardized benchmarks for code completion approaches.

Chapter 5 discusses our methodology in detail. Besides, the obtained evaluation results are used as our response to the third core research question (CRQ3). Ultimately, to facilitate the reproducibility, all the related datasets and source code of our proposed approach have been made accessible to the community.

An approach to integrate user feedback into code generation models without explicit re-training

To tackle the last core research question (CRQ4), we proposed a methodology named *One-shot Correction* to integrate user feedback into generative AI models without re-training, while supporting thorough analysis of unexpected outcomes (Chapter 6). The former is achieved by (i) an additional correction data-store to accumulate feedback from users, (ii) k-Nearest Neighbor approach to retrieve the correction information across sessions, and (iii) decomposition techniques to divide code generation into sub-problems. As a result, the provenance of translated code is audited and therefore enables the latter, i.e. intensive inspection of incorrect outcomes.

We employed a prototype of *One-shot Correction* and conducted an extensive comparison between code produced by GPT-3.5[10] and by our prototype to illustrate the utility of the proposed method. Additionally, to exhibit the benefit of using *One-shot Correction* in customizing and verifying suggested code snippets, we developed a preliminary Graphical User Interface (GUI) that translates an NL input query to Python code, using the proposed concept. All the source code, utilized test suites, and evaluation results are made publicly available.

1.6 Publications

This doctoral project comprises the following practical and research publications:

1. A Visual Studio Code extension named *NLDSL*

On the practical side of the project, we deployed the *NLDSL* extension on 13 July 2020 and keep maintaining it since then. As mentioned in Section 1.5, the extension generates code completions for external DSLs and translates DSL commands to executable code snippets. Further insights regarding the design, dissemination, and preliminary evaluation of the extension, along with an answer to the first core research question (CRQ1) are discussed in Chapter 3. Besides, details of contributors are outlined in Section 3.6 of that chapter.

For the research contributions, we published the following papers:

2. Schumacher, M.E.H., Le, K.T., and Andrzejak, A. (2020). "Improving Code Recommendations by Combining Neural and Classical Machine Learning Approaches", in *Proceedings*

[10] Models from OpenAI, https://platform.openai.com/docs/models/gpt-3-5, (Accessed: 08 December 2023).

of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pp. 476-482, DOI: https://doi.org/10.1145/3387940.3391489.

This paper demonstrates the potential of ensemble-like methods for code completion and recommendation tasks in dynamically typed languages. A thorough evaluation of the *Extended Network model* is illustrated and explained in the paper. The model is proposed based on approaches of classical and neural ML. Overall, the model is able to (moderately) surpass its components. Chapter 4 reiterates the evaluation from this paper and discusses the results related to addressing the second core research question (CRQ2).

Contribution of authors: Schumacher, M.E.H. proposed the *Extended Network model*. Schumacher, M.E.H. and Le, K.T. wrote the manuscript, performed model evaluation, and analyzed the results with support from Andrzejak, A. All authors discussed the results. Andrzejak, A. supervised the project.

3. Le, K.T., Rashidi, G., and Andrzejak, A. (2021). "A Methodology for Refined Evaluation of ML-based Code Completion Approaches". In *KDD Workshop on Programming Language Processing (PLP)*.

4. Le, K.T., Rashidi, G., and Andrzejak, A. (2023). "A Methodology for Refined Evaluation of Neural Code Completion Approaches". *In Special Issue on Programming Language Processing, Data Mining and Knowledge Discovery*, 37(1), pp. 167–204, DOI: https://doi.org/10.1007/s10618-022-00866-9.

Paper (4) is the journal version of paper (3). Both papers present a methodology named *Code Token Type Taxonomy (CT3)*, which is proposed for a refined evaluation of ML-based code completion models. The utility of the proposed methodology is demonstrated by performing an empirical study on a Transformer-based code completion approach.

The journal version enhances paper (3) by surveying state-of-the-art ML-based code completion models (from 2018 to 2021) and modifying the *CT3* schema to support distinguishing between definitions and usages of identifiers. Information of additional experiments related to utilized thresholds is also provided. Chapter 5 rehashes the *CT3* methodology in the journal version while exploring the findings relevant to tackling the third core research question (CRQ3).

Contribution of authors: Le, K.T. wrote the manuscript (3) with support from Rashidi, G., and Andrzejak, A., performed the empirical study on the Transformer-based model, analyzed the results, conducted the state-of-the-art survey, modified the *CT3* schema, deployed additional experiments, updated the results, and subsequently completed the manuscript for the journal version (4). Rashidi, G. developed the original *CT3* schema. All authors discussed the results. Andrzejak, A. supervised the project.

5. Le, K.T. and Andrzejak, A. (2024). "Rethinking AI Code Generation: A One-shot Correction Approach Based on User Feedback". In *Automated Software Engineering*, 31(60), DOI: https://doi.org/10.1007/s10515-024-00451-y.

This work introduces a methodology named *One-shot Correction* to incorporate user feedback into generative AI models without re-training. The method also enables thorough examination of unexpected results through straightforward approaches and facilitates insights for potential improvements. Additionally, a preliminary GUI application was developed to demonstrate the utility of our approaches in simplifying customization and assessment of suggested code for users. Chapter 6 restates this paper's content and includes a response to the final core research question (CRQ4).

Contribution of authors: Le, K.T. proposed the *One-shot Correction* methodology with support from Andrzejak, A., implemented the method, conducted extensive evaluation and ablation study, analyzed the results, developed the GUI, and wrote the manuscript. All authors discussed the results. Andrzejak, A. supervised the project.

1.7 Structure of the Dissertation

The dissertation is organized into chapters, sections and subsections, adhering a specific format.

Content Distribution

The dissertation is structured in five parts and 8 main chapters. Part I (Chapters 1–2) introduces the thesis, alongside the knowledge background and related work. Part II (Chapter 3) presents the practical side of the thesis while Part III (Chapters 4–6) describes the research contributions. Part IV (Chapters 7–8) concludes the thesis and discusses potential future work. Ultimately, we group the lists of acronyms and references in Part V. Furthermore, supplementary information that complements the background knowledge can be found in Appendices A and B.

Table 1.1 exhibits a mapping of chapters, target problems (Section 1.1), and core research questions (Section 1.4) addressed in the dissertation.

Table 1.1: A mapping from chapters to target problems and core research questions (CRQs).

Chapter(s)	**Tackled problems**			**Resolved CRQs**
	Programming barrier </>	*Reuse problem*	*Scalability problem*	
1, 2	–	–	–	–
3	●	●	●	CRQ1
4	●	○	○	CRQ2
5	●	◐	◐	CRQ3
6	●	◐	○	CRQ4
7, 8	–	–	–	–

– The chapters only provide introductions/conclusions for the issues/questions.
● The issue is the primary objective of the chapter.
◐ Methods presented in the chapter provide a part of the solution for the issue.
○ The issue it not the main concern of the chapter.

Specifically, each chapter conveys content as follows:

- Chapter 2 provides background knowledge and related work on two aspects central to our research: facilitating programming in data analysis tasks through *low-code/no-code* development and expediting programming with ML-based methods. Additionally, this chapter includes a dedicated section on the Transformer architecture, pivotal in numerous recent generative AI models.
- Chapter 3 outlines practical contributions to the dissertation, focusing on accelerating programming through *low-code* approaches by aiding DSL development. It includes an introduction to the vanilla *NLDSL* tool and a detailed discussion of enhancements to the *NLDSL* extension, as well as our dissemination strategy.
- Chapter 4 introduces and assesses *Extended Network model*, an ensemble-like model designed to enable code recommendation in dynamically typed languages. Motivation, related work, the concept with an illustrative model, and a comprehensive evaluation for the proposed ensemble-like architecture are respectively discussed in this chapter.
- Chapter 5 demonstrates utility of *CT3* methodology for refined evaluation of ML-based code completion approaches. Similar to Chapter 4, this chapter also presents motivation, background, the methodology, and evaluation results of the approach.
- Chapter 6 illustrates the *One-shot Correction* methodology to integrate user feedback into generative AI models without retraining. The evaluation results highlight the ability of the proposed method in enabling thorough examination on unexpected outcomes. The preliminary GUI application is also revealed in this chapter.
- Ultimately, Chapter 7 summarizes our contributions, followed by potential further research directions delineated in Chapter 8.

Formatting Convention

To aid readers in quickly grasping the ideas of each chapter, in addition to headings, we employed a formatting convention for content structuring within a section as follows:

Color box type 1:

> Texts in this layout deliver important points or findings, which will be further clarified in the immediately following paragraph(s).

Color box type 2:

> **This configuration is utilized for key contents of a section, namely core research questions, contributions, and responses to the research questions.**

Unlike the first type of color box, which is always placed before the detailed explanation, the second type can be positioned either at the start (i.e. presenting core research questions or

contributions) or at the end of a section (i.e. recapping responses to a research question defined within a chapter or a core research question of the dissertation).

Additionally, multiple predefined icons are integrated throughout the dissertation to highlight corresponding concepts or to categorize relevant ideas. These icons are intended to expedite readers in skimming the text. Table 1.2 provides a concise overview of the utilized icons.

Table 1.2: A summary of predefined icons used in the dissertation.

Purpose	**Icon**	**Meaning**
Concept labeling		Programming barrier
		Reuse problem
		Scalability problem
Information marking		Advantage
		Disadvantage
		Outlook or suggestion for the above disadvantage
		Notable tool-kit
		Application field

Chapter 2
Background and Related Work

Although data science-related jobs are considered among the most rapidly-growing worldwide, data scientists still devote more than a third of their time to data cleansing and preparation. Moreover, researchers and practitioners must engage a plethora of Programming Languages (PLs) and data science tool-kits, including the impediment of transitioning between languages during model deployment (more details in Appendix A).

Our work targets to accelerate the code-centric tasks of these jobs, for both data scientists and practitioners. We pursue this objective through two directions, partitioning the contributions of our work into practical and research sides, namely: (i) facilitating data analysis and processing tasks in *low-code* manner and (ii) advancing programming with Machine Learning (ML)-based approaches. Before delving into our individual contributions, this chapter provides background knowledge and relevant works, serving as foundation for our proposed methodologies.

Particularly, Section 2.1 introduces an overview of *low-code* and *no-code* methods, discusses advantages and challenges of Domain-Specific Languages (DSLs) in accelerating programming, and highlights prominent data analysis and processing tool-kits that employ these techniques. Subsequently, Section 2.2 outlines well-known Machine Learning (ML)-based approaches for augmenting general programming tasks. Furthermore, due to the importance of Transformer model, the power behind numerous notable generative AI models in recent years, we dedicate Section 2.3 to presenting the model's architecture and reviewing its successors. Ultimately, we end this chapter by a brief summary in Section 2.4.

2.1 Facilitating Data Analysis and Processing in Low-code Manner

As mentioned above, data analysis tasks involve multiple tool-kits and programming languages. This requires specialized skill-sets, including the ability to code for data analysts. However, with the increasing significance of data, the need for broader access to data analysis and processing beyond specialists emerges as a pressing concern.

Consequently, *low-code* to *no-code* applications are deployed to enable users of all experience levels to analyze data with minimal coding skills (Karl, 2023). This section firstly highlights the pros and cons of *low-code* and *no-code* approaches, then introduces the backbone of these techniques, Domain-Specific Languages (DSLs). Ultimately, well-known tool-kits in *low-code* or *no-code* fashion for data analysis and processing tasks are outlined to conclude the section.

2.1.1 Low-code and No-code Development

Forrester[1] presented the term "low-code" to the public in 2014, referring to *"platforms that enable rapid application delivery with a minimum of hand-coding, and quick setup and deployment, for systems of engagement"* (Richardson et al., 2014; Luo et al., 2021). Despite the longstanding presence of the idea, the market of these platform-based development approaches has only begun to grow exponentially in recent years (Luo et al., 2021; Elshan et al., 2023; Hirzel, 2023).

An Overview of Low-code and No-code Development

> *Low-code*: Low effort of coding in general-purpose programming languages (e.g. Python).

There are various names adopted to refer to such techniques, such as *Low-code Development* (Luo et al., 2021), *Low-code Programming* (Hirzel, 2023), *Low-code Platform*, *Low-code Development Platform*, and *Low-code Application Platform* (Bock et al., 2021; Elshan et al., 2023). Regarding the terms used, the conceptual goal is to allow users to develop applications with less coding effort by minimizing the use of GPLs and instead utilizing alternative means such as Graphical User Interface (GUI) or Natural Languages (NLs) (Luo et al., 2021; Elshan et al., 2023; Hirzel, 2023).

Notable examples of Low-code Application Platforms comprise ⚒ OutSystems[2], ⚒ Mendix[3], and ⚒ Microsoft Power Apps[4]. Prominent advanced *low-code* features supported by these tools involve AI-augmented development, third-party Continuous Integration (CI) and Continuous Deployment (CD), on-premises deployments, multiexperience development, and publishing a custom AI model (Vincent et al., 2022). However, each product has its own strengths and constraints which should be thoroughly considered before deploying. Discussing these platforms in detail is beyond the scope of this dissertation.

> Unlike *low-code*, *no-code* requires no coding experience from users and therefore is more suitable for end-users than professional developers.

While Luo et al. (2021) and Bock et al. (2021) considered these terms as synonyms, Hirzel (2023) and Elshan et al. (2023) argued that there is a slight difference between these development approaches. In the second view, *no-code* possesses significant overlap with *low-code* while requiring zero handwritten code in a general-purpose programming language (Hirzel, 2023). The latter is accomplished through GUIs, enabling users to manipulate input and output by selecting and arranging (or by *drag and drop* in some applications), configuring, and connecting elements from built-in program libraries or third-party plugins (Elshan et al., 2023).

Examples of *no-code* platforms include website builder tools, such as ⚒ Shopify[5], ⚒ Wix[6],

[1] A research and advisory company, https://www.forrester.com.
[2] Low-code Development Platform, https://www.outsystems.com/.
[3] Low-code Platform, https://www.mendix.com/.
[4] Build Apps with AI, https://www.microsoft.com/en-us/power-platform/products/power-apps.
[5] Commerce platform, https://www.shopify.com/.
[6] Website builder, https://www.wix.com/.
(Accessed: 02 January, 2024).

and ⚒ WordPress[7]. On these platforms, users can drag and drop provided components to build their websites with minimal knowledge of web development. Features of the websites can be expanded by installing third-party plugins, for instance Search Engine Optimization (SEO) plugins for commerce sites. Users can also write small scripts to customize some components in their websites (considered as *low-code* cases).

However, supported elements in *no-code* platforms are restricted to complex features of back-end applications and not designed for professional developers. This arises from the predefined construction of these elements and the lack of accompanying scripting languages (Hirzel, 2023; Elshan et al., 2023). Additionally, Luo et al. (2021) indicated in their surveys that *low-code* and even *no-code* techniques do not exclude the use of GPLs, particularly for complicated issues. Therefore, ***we consider* no-code *as a sub-domain of* low-code *and cover its concept in the following when using the term "low-code".***

Low-code Development Usage: Who and Where

Hirzel (2023) pointed out that users of *low-code* range from professional developers to so-called *citizen developers*. The latter are users with little or no formal programming skills but tend to have more extensive domain knowledge. Domain experts in explicit disciplines such medicine, chemistry, and engineering become citizen developers with the aid from *low-code* approaches. These platforms concurrently amplify professional developers' productivity by expediting tedious procedures, allowing them to concentrate on advanced tasks.

Studies published by Luo et al. (2021), Johannessen et al. (2021), and Di Ruscio et al. (2022) reveal that *low-code* techniques were employed on various application types (e.g. mobile, web) across multiple application domains (e.g. E-Commerce, Business Process Management, and Social Media). Specifically, *low-code* brings the most benefits for repetitive and time-consuming tasks, facilitating faster completion for developers. Besides, *low-code* is also suitable for rule-based operations with minor exceptions, given its limitation in customization (Hirzel, 2023).

Benefits, Limitations and Outlook of Low-code Development

A survey conducted by Luo et al. (2021) highlights various 👍 advantages and 👎 disadvantages of *low-code* approaches, cited by developers through online communities. Other studies from Elshan et al. (2023), Hirzel (2023), and Martinez et al. (2023) reaffirm the points. Besides, these authors also provided their ℹ perspectives on mitigating the identified issues.

> 👍 *Low-code* serves as a shared language, enhancing communication between professional and citizen developers by providing common perspectives.

Apart from the target to 👍 minimize coding effort, *low-code* platforms can 👍 accelerate development, and in some cases, 👍 reduce organizational costs. This is attributed to the fact that less technically savvy employees are enabled to implement their application ideas, alleviating the workload and demand for programming experts. Besides, most *low-code* applications come

[7]Web content management system, https://wordpress.com/, (Accessed: 02 January, 2024).

with GUIs to 👍 offer a newbie-friendly and easy way for citizen developers to get familiar with the apps, compared to learning a new general-purpose programming language (Luo et al., 2021).

Notably, both Hirzel (2023) and Elshan et al. (2023) agree that *low-code* techniques can be utilized to 👍 communicate among users, specifically to mitigate the tension between professional developers and domain experts. For instance, end-users or domain experts might use *low-code* tools to illustrate their desired operations, while professional developers explain existing functions or propose a new feature in a user-friendly way through these tools.

> 👎 Significant drawbacks of *low-code* platforms include the dependency on vendors (i.e. *vendor lock-in*) and the lack of source code access.

Low-code applications not only offer undeniable rewards but also entail inherent shortcomings. Respondents from the surveys of Luo et al. (2021), Elshan et al. (2023), and Martinez et al. (2023) emphasized 👎 the reliance on specific vendor's environments while using *low-code* applications, or so-called *vendor lock-in*. Users are bound to the utilized platform, obligating them to accept vendor-imposed changes, such as price escalation or service termination.

A contributing factor to the above drawback is 👎 the retention of users' source code by most *low-code* platform vendors (Elshan et al., 2023). In other words, 👎 users are hindered while transitioning between platforms. Besides, no access to source code also causes 👎 restrictions in maintenance and debugging capabilities, 👎 impeding experienced developers from performing their tasks (Luo et al., 2021).

Furthermore, *low-code* is less powerful than traditional programming due to 👎 the lack of customization, particularly for complicated components. In addition, while *low-code* platforms may reduce the need for professional developers, they encounter 👎 the issue of raising licensing costs with increased number of users (Luo et al., 2021; Martinez et al., 2023).

Ultimately, by exploring prominent techniques of *low-code*, Hirzel (2023) identified further weaknesses. For instance, although Programming by Natural Language (PBNL) enables users to communicate with applications through NL, 👎 the generated programs from these NL queries are often wrong due to the ambiguity of NL.

> ℹ️ Leveraging Artifical Intelligence (AI) and exposing Domain-Specific Languages (DSLs), the cornerstone of *low-code* techniques, can address the previously outlined disadvantages.

To prevent or alleviate the aforementioned problems, developers are recommended to carefully assess a *low-code* platform before integrating it into their business. For example, ℹ️ open-source platforms might mitigate the concerns related to *vendor lock-in* and missing source code access. Meanwhile, ℹ️ ensuring that the platform provides sufficient implementation units with flexibility might lessen the issue of lacking customization (Luo et al., 2021).

Besides, ℹ️ AI can be used to enhance *low-code* techniques, e.g. by deploying Large Language Models (LLMs) in processing NL and generating prominent code snippets (Hirzel, 2023; Cai et al., 2023). More details about these advancements are presented in Section 2.2. Notably, Hirzel (2023) also underlined in his study that most *low-code* techniques inherently correlated with

DSLs, i.e. programming languages tailored to specific domains (Mernik et al., 2005). However, these associated DSLs are not always accessible to users. Hirzel argued that by exposing DSLs for *low-code*, users can test, audit, and share them across applications, tackling the obstacles of *vendor lock-in* and transition between platforms.

It is worth noting that the viewpoints given by Elshan et al. (2023) and Martinez et al. (2023) mostly focus on the business side and therefore are excluded from this discussion. The concept, benefit, and limitation of *low-code* consolidate our motivation behind the published *NLDSL* extension (Chapter 3). In the next subsection, we introduce an overview of DSLs, their advantages and challenges, serving as background for this contribution.

2.1.2 Domain-Specific Languages

Domain-Specific Languages (DSLs) serve as the backbone of *low-code* techniques, sharing the goal of expanding opportunities for domain experts to express and implement their requirements. Traditionally, domain experts with specific knowledge must describe their needs to developers, who encode the domain knowledge in GPLs like Python or Java. This process results in cryptic scripts that are challenging for domain experts. Besides, implementing individual requirements is error-prone and laborious. Moreover, the resulting code obscures the core knowledge, preventing domain experts from recovering the original business logic (JetBrains, 2017).

Additionally, communication between domain experts and developers is acknowledged as the most challenging aspect of software projects and a prevalent cause of project failure (Fowler, 2010). Consequently, this issue underscores the need for a specialized programming language with a higher level of abstraction than GPLs, facilitating better understanding and utilization by domain experts. DSLs fulfill this requirement.

It is worth noting that DSL is not a recent research concept. Indeed, in the 1980s, DSLs were referred by various names such as special purpose, task-specific, application or *little* languages (Mernik et al., 2005). These names already convey the essential properties of such languages.

An Overview of DSLs

> DSL: A highly abstracted programming language tailored for a particular domain.

It is important to firstly clarify that we avoid using "expressiveness" in the definition as proposed by Mernik et al. (2005) and Fowler (2010) due to their distinct interpretations. Mernik et al. (2005) associated the word with the state of showing thoughts[8], i.e. the ease of conveying business logic in this case, asserting that GPLs are less expressive than DSLs. Meanwhile, Fowler (2010) referred to the breadth of capabilities supported by a language (Felleisen, 1990), placing GPLs above DSLs in "expressiveness". The latter is shaped by GPLs' adaptability across diverse problems, solutions, or businesses, whereas DSLs offer essential features within their domains.

Inspired by Fowler (2010) and to minimize ambiguity, we adjust the definition for the four key elements of a DSL as follows:

[8] Cambridge's definition of "expressiveness", https://dictionary.cambridge.org/dictionary/english/expressiveness, (Accessed: 07 January 2024).

- *Computer programming language.* A DSL is indeed a programming language, designed in an ease of use manner for humans. Although Mernik et al. (2005) and Managoli (2020) noted that DSLs can be non-executable, Fowler (2010) argued for their executability. This dissertation targets DSLs of the latter case.

- *Language nature.* A DSL is a language where fluency arises not only from individual expressions but also from their composability.

- *High level of abstraction.* As mentioned above, a DSL is expected to provide higher abstraction level than GPLs, aiding domain experts in comprehending and implementing the code written in this language.

- *Domain focus.* Unlike GPLs, DSLs address problems in a specific field of expertise, e.g. HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) for web application, Structured Query Language (SQL) for database queries, or Extensible Markup Language (XML) for markup languages (Mernik et al., 2005; Managoli, 2020).

> There are two types of DSLs: *internal* and *external* DSLs, distinguished by their underlying parsing mechanisms.

Particularly, *internal* and *external* DSLs are classified based on the differences between two following language types, according to Fowler (2010) and Managoli (2020):

- *DSL script*: The language used for writing or presenting a DSL.

- *Host language*: The GPL in which the DSL is processed and executed.

Internal DSLs. An *internal* DSL is a special way of using GPLs, creating the impression of a custom language within the *host language* through libraries written in that *host language* (Fowler, 2010; Korz et al., 2023). *DSL script* and the *host language* are identical in this case. Therefore, *internal* DSLs is also known as *embedded* DSLs (Kosar et al., 2008; Managoli, 2020).

To exemplify an instance of *internal* DSLs, Sanaulla (2013) followed the guidelines of Fowler (2010) and implemented his DSL in Java using *Method Chaining*. For simplicity, we use an Unified Modeling Language (UML) class diagram to depict his code in Figure 2.1. Listing 2.1 demonstrates the code for graph creation using the implemented DSL, while Listing 2.2 displays the code without utilizing the DSL, achieving the same task.

Initially, three classes `Graph`, `Edge`, and `Vertex` (Figure 2.1) are created as basic elements. A graph can encompass multiple edges and vertices, with each edge being defined by two vertices and a specific weight. Without further implementation, a graph can be constructed using these three classes as illustrated in Listing 2.2.

The *internal* DSL of Sanaulla is implemented via two additional classes, namely `GraphBuilder` and `EdgeBuilder`. Methods provided by these classes (e.g. method `edge` of class `GraphBuilder`, methods `from`, `to`, and `weight` of class `EdgeBuilder`) facilitate graph building in a concise way, chaining methods one after another, as represented in Listing 2.1. Fowler (2010) called the DSL

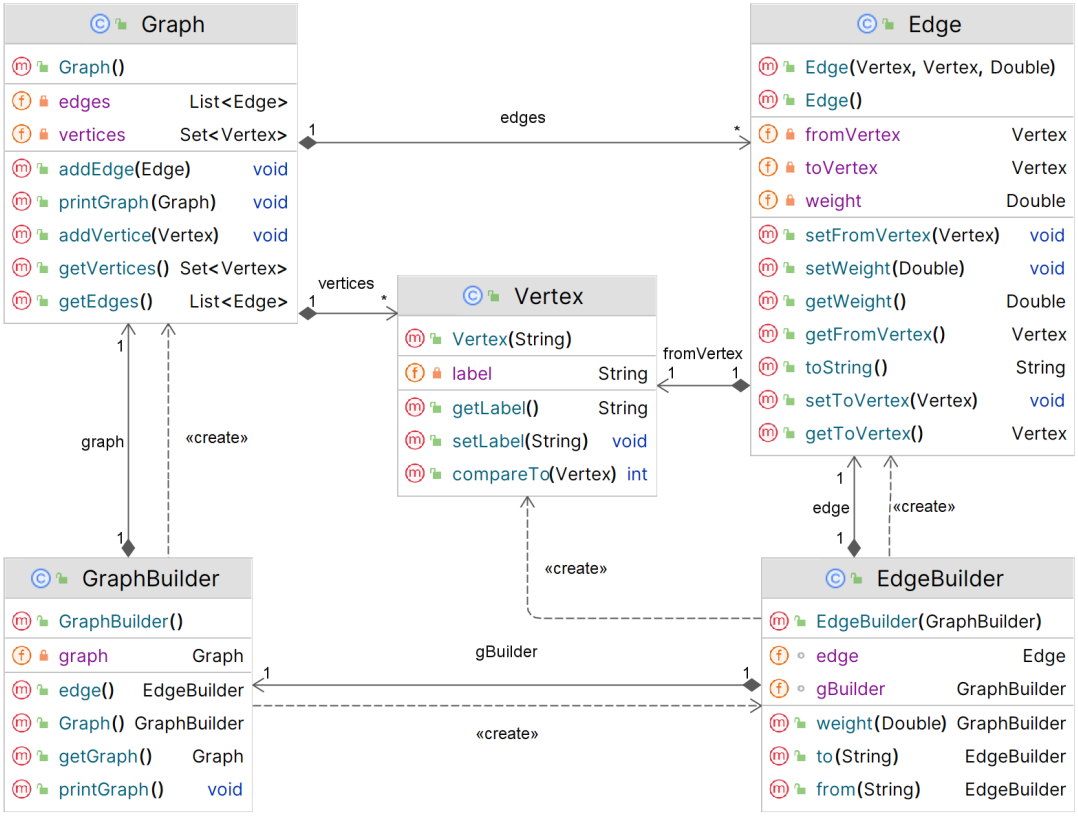


Figure 2.1: UML class diagram* of Java code used to create an *internal* DSL (Sanaulla, 2013).

* The class diagram was created using IntelliJ IDEA Ultimate.

```
Graph()
  .edge()
    .from("a")
    .to("b")
    .weight(10.0)
  .edge()
    .from("b")
    .to("c")
    .weight(20.0)
  .edge()
    .from("d")
    .to("e")
    .weight(50.5)
  ...

```

Listing 2.1: Creating a graph with the *internal* DSL, i.e. methods from classes `GraphBuilder` and `EdgeBuilder`.

```
Vertex vA = Vertex("a");
Vertex vB = Vertex("b");
Vertex vC = Vertex("c");
...
Edge abEdge = Edge(vA, vB, 10.0);
Edge bcEdge = Edge(vA, vB, 20.0);
...
Graph sampleGraph = Graph();

sampleGraph.addVertice(vA);
sampleGraph.addVertice(vB);
...
sampleGraph.addEdge(abEdge);
sampleGraph.addEdge(bcEdge);
...
```

Listing 2.2: Creating a graph without the *internal* DSL, i.e. excluding classes `GraphBuilder` and `EdgeBuilder`.

designed in this such case as *fluent interface*, resulting from its English-like readability. We prefer to the code of Sanaulla (2013) for the detailed implementation.

External DSLs. In contrast to *internal* DSLs, the language used to present an *external* DSL (i.e. *DSL script*) and its *host language* are clearly distinguishable. An *external* DSL usually has a custom syntax or adopts a syntax from another language. Popular examples of such *external* DSLs are regular expressions, SQL and XML (Fowler, 2010).

For instance, referencing the time allocation survey conducted by Anaconda (2022) illustrated in Table 2.1, users can use *external* DSLs to identify the first three tasks occupying more than 20% of data scientists' working time. Listing 2.3 and 2.4 display the corresponding commands written in SQL and in our *NLDSL* (Chapter 3), respectively.

Table 2.1: A survey of how data scientists spend their time (Anaconda, 2022).

Tasks	Duration(%)*
Data prep. & clean.	38
Model selecting	9
Model training	9
Model deploying	9
Data visualization	13
Reporting & presentation	16
Other	7

* Due to rounding, the total is not 100%.

Question: Which three tasks consume more than 20% of data scientists' working time?

```
SELECT TOP 3 tasks
FROM table_survey
WHERE duration > 20;
```

Listing 2.3: SQL query.

```
on df_survey | select rows df_survey. ↩
    ↪ duration > 20 | select columns   ↩
    ↪ "tasks" | head 3
```

Listing 2.4: *NLDSL* statement.

Both SQL and *NLDSL* employ distinct syntaxes. Queries in SQL are separated by semicolons and can span one or multiple lines. In contrast, *NLDSL* breaks a one-line statement into operations connected by vertical dashes. Additionally, Python serves as the *host language* for *NLDSL*, while SQL can be processed and executed by various *host languages*, depending on its version. For example, C/C++ is utilized for MySQL (Pachev, 2007).

In contrast to *internal* DSLs, which necessitate knowledge of the implementing language, *external* DSLs are more suitable to reduce programming complexity for domain experts. As a result, we deploy a tool utilizing our proposed *external* DSL, primarily for data analysis and processing tasks, to illustrate our practical contribution in Chapter 3.

DSL Usage: Who and Where

Developers and domain experts can leverage DSLs to fulfill specific objectives. For instance, in web construction, DSLs such as HTML and CSS find application for both developers and domain experts (Fowler, 2010). However, in contrast to approaches like *low-code* or *no-code*, DSLs still operate as programming languages, necessitating domain experts to possess a certain level of proficiency in the languages involved. For developers, several DSLs can also generate corresponding scripts in GPLs for them (Córdoba-Sánchez et al., 2016; Andrzejak et al., 2019a; Dejanović et al., 2021; Korz et al., 2023).

Applications of DSLs span across diverse domains, ranging from programming (Córdoba-Sánchez et al., 2016), data analysis (Andrzejak et al., 2019a; Korz et al., 2023), smart contract (Wöhrer et al., 2020), robotics (Dhouib et al., 2012) to life science (Lakin et al., 2020) and even psychology in social science (Dejanović et al., 2021). Despite varying research fields, these DSLs collectively aim to address specific issues within their respective domains. However, developing DSLs poses certain challenges, constituting inherent limitations of these languages.

Benefits and Limitations of DSLs

Endowed with the aforementioned design attributes, DSLs offer numerous 👍 advantages for developers and domain experts. However, their benefits also come with associated 👎 costs.

> 👍 DSLs can facilitate the reuse of software artifacts, while advancing communication among developers and domain experts.

Similar to *low-code*, DSLs make software development more sufficient by multiple means. Firstly, 👍 the clarity delivered by DSLs improves code readability, understanding, error identification, and modification, thereby enhancing development productivity. Moreover, the restricted scope and structure of DSLs make incorrect usage more challenging (Fowler, 2010).

Secondly, 👍 DSLs expedite the reuse of software artifacts, such as source code and designs. Prominent instances involve *internal* DSLs, implemented as APIs within their *host languages* (Mernik et al., 2005) and *external* DSLs capable of generating GPLs for users. It is important to recall here that the *reuse problem* 🔄 is one of the key focuses of this dissertation.

Ultimately, 👍 through clear and precise language, DSLs establish a shared understanding for both domain experts and developers, hence alleviating the inherent communication challenges between these users (Fowler, 2010; Andrzejak et al., 2019a). DSLs enable developers to create an environment where domain experts can express requirements in an intuitive manner, preserving business logic without burdening experts with intricate implementation details (JetBrains, 2017).

> 👎 DSL development is challenging, requiring expertise in both domain knowledge and language development.

Mernik et al. (2005) and Alves (2023) emphasized that 👎 developing DSLs expects substantial effort and thorough consideration of various factors from business to technical aspects, such as user community size, training, support, and maintenance overheads. Besides, 👎 constructing a functional DSL demands expertise in both domain knowledge and language development, a skill set found in only a limited number of individuals.

Additionally, drawbacks for each type of DSL are also underlined by Fowler (2010) and Korz et al. (2023). 👎 *Internal* DSLs are constrained by the *host language*, resulting in inflexible syntaxes and support. Meanwhile, 👎 *external* DSLs offer diverse syntaxes but involve high development costs, spanning from parsers to code generators or compilers. Moreover, 👎 *external* DSLs might encounter challenges in efficiently communicating with other languages used in a project.

Language Workbenches for DSLs

> *Language workbenches*: Software engineering tools that streamline the creation and modification of programming languages through advanced editors, significantly reducing the development cost.

Fowler (2010) introduced the term *language workbenches* specifically for DSLs. This term refers to tools that assist users in defining and editing DSLs, making it more feasible for domain experts to write their own DSLs. Campagne (2014) and Erdweg et al. (2015) extended the definition to computer languages in general.

Language workbenches typically support users in defining three following aspects of a DSL environment (Fowler, 2010):

- The language schema (e.g. data structure) of the DSL.
- The editing environment, i.e. the editor, where users can write and modify the DSL, including textual editor and projectional editor. While the former assists modification with text only, the latter expands its format to diagrams, tables, or forms.
- The functionality of the DSL script, often involving code generation.

Notable *language workbenches* for DSLs include Spoofax (Kats et al., 2010), MPS (Campagne, 2014), Xtext (Bettini, 2016), and textX (Dejanović et al., 2017). MPS uniquely features a projectional editor, while textX is specifically employed for Python-based DSL creation, distinct from the Java-centric approach of the other tools. Additionally, all these tools facilitate the generation of DSLs into corresponding GPLs (e.g. Java or Python).

Erdweg et al. (2015) defined a feature model outlining essential characteristics of a *language workbench* and systematically evaluated ten prominent tools by using them to develop a new questionnaire DSL. The study found that all utilized language workbenches met the fundamental criteria for rendering and executing the new DSL. Furthermore, the authors observed a marked improvement compared to manual implementation.

In summary, DSLs with *language workbenches* and *low-code* approaches provide unequivocal benefits in enhancing the capabilities of domain experts, enabling task accomplishment without extensive technical expertise. In the next subsection, we give an overview of notable *low-code* tool-kits deployed for expediting data analysis and processing tasks.

2.1.3 Low-code Tool-kits for Data Analysis and Processing

Multiple *low-code* research directions have emerged to accelerate and automate data analysis and processing tasks while enhancing intuitiveness. Theses fields include mixed-initiative approaches (Makonin et al., 2016), predictive interaction strategies (Heer et al., 2015), data processing based on input-output examples (Gulwani, 2011), and a fusion of live programming with GUIs and DSLs (DeLine, 2021). The shared foundational concept of these methods is illustrated in Figure 2.2. The notion encompasses (1) a predictive system learning user intention for guiding

potential subsequent data handling steps, ❷ human analytical reasoning in selecting appropriate actions, and ❸ a real-time data visualization interface responding to user-initiated changes.

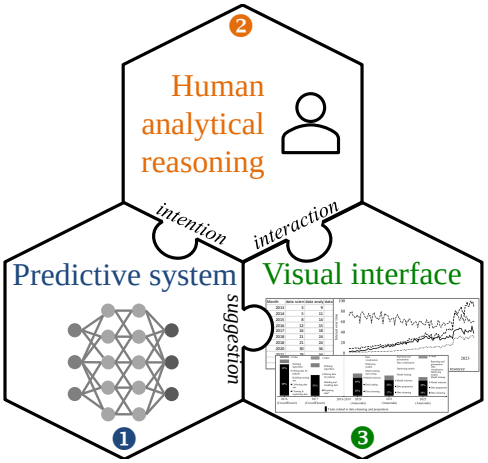


Figure 2.2: Underlying concept of *low-code* approaches for data analysis and processing, adapted from Figure 1 of Makonin et al. (2016).

Data Wrangling ⚒ Tool-kits in Research

> A majority of researchers sought to tackle the data transformation challenge, commonly referred to as *data wrangling*. In this case, the predictive system of the data wrangling tool recommends suitable actions to handle diverse data formats from various sources and standardize the data for compatibility with further analysis tools (Raza et al., 2017).

One prevalent approach for suggesting potential operations is based on a set of implemented ones. Notable tools in this domain include ⚒ Wrangler Trifacta (Kandel et al., 2011), ⚒ *NLDSL* (Andrzejak et al., 2019a), and ⚒ Virtual DSL (Korz et al., 2023). While Wrangler Trifacta facilitates interaction through a data table with a range of possible actions, the other two tools incorporate DSLs into source code files. This enables users to utilize and define their own functions, seamlessly transitioning between DSLs and GPLs. Detailed discussion on *NLDSL* is presented in Chapter 3.

Another strategy for prompting next possible steps involves utilizing input-output examples, where users provide pairs of input data and corresponding target output. This enables the predictive system to learn user intention from these examples and apply similar actions to the remaining data. ⚒ FlashFill (Gulwani, 2011), a popular example featured in Microsoft Excel, employs this method. While ⚒ WREX (Drosos et al., 2020) and ⚒ FlashFill++ (Cambronero et al., 2023) adhere to a similar principle, Raza et al. (2017) adjusted the underlying concept and proposed a ⚒ predictive program synthesis technique. This procedure infers data extraction actions based solely on input examples, exempting users from the obligation to explicitly specify their purpose.

Notably, AI can be used to augment the predictive system. Petricek et al. (2022) pointed out that employing an AI assistant framework for semi-automated data wrangling, guided by users, would streamline data processing tasks compared to manual or fully automatic tools.

Furthermore, Jaimovitch-López et al. (2023) exploited data wrangling using ⚒ GPT-3 DaVinci, a generative AI model, with other tools such as FlashFill and Wrangler Trifacta, obtaining competitive results.

Ultimately, Shrestha et al. (2021) developed ⚒ Unravel, a *low-code* tool designed to assist data scientists in detailed code comprehension and exploration. Instead of predicting next actions, the underlying system of the tool dissects data wrangling code into multiple transformations. Users can inspect, enable/disable each transformation, or rearrange the order via drag-and-drop. This leverages the design pattern known as *fluent interface*, where the code represents a single chain of various operations on a data table (similar to the example in Listing 2.1).

Data Preparation ⚒ Tool-kits in Industry

> In industry, *low-code* data preparation and analytics tools provide device-agnostic and platform-independent support. These tools reach beyond the analysis and processing tasks, toward advanced features such as report generation and cloud-based collaboration.

Dearmer (2023), Agadumo (2023), and Haan (2024) revealed recent top data preparation and analytics tools, highlighting key features of each tool. They also pointed out that selecting the appropriate tool involves comprehensive assessment of multiple factors such as ease of use, supported features, compatibility, collaboration, scalability, and last but not least pricing.

For instance, ⚒ Microsoft Power BI[9] (i.e. Business Intelligence) is recommended for small organizations, whereas ⚒ SAP Analytics Cloud[10] might be cost-prohibitive for small-scale enterprises. Besides, while most tools support both desktop and mobile applications, some, such as ⚒ Alteryx[11] (powered by Trifacta), ⚒ Altair Monarch[12], ⚒ KNIME[13], and ⚒ Datameer[14], primarily focus on desktop users.

In addition, these tools offer different *low-code* methods to process data. Alteryx, KNIME, Datameer, and ⚒ Tableau[15] enable users to define preparation workflows through a chain of built-in components. Meanwhile, Microsoft Power BI, SAP Analytics Cloud, and Altair Monarch communicate business insights through interactive data visualizations (e.g. charts and graphs).

Notably, all these tools increasingly integrate AI into their systems, utilizing it for tasks such as discovering data patterns, generating reports, and automating repetitive content creation tasks. An exemplar in this domain is ⚒ DataGPT[16], introduced in October 2023 as the first conversational AI data analytics software (Rubin, 2023). Powered by GPT-4, a generative AI model developed by OpenAI, DataGPT uniquely engages with users through English dialogues, exempting them from programming tasks.

[9] Data visualization software, https://www.microsoft.com/en-us/power-platform/products/power-bi.
[10] All-in-one cloud product, https://www.sap.com/products/technology-platform/cloud-analytics.html.
[11] Data science and analytics automation platform, https://www.alteryx.com/.
[12] Self-service data preparation solution, https://altair.com/monarch.
[13] Open source platform, https://www.knime.com/.
[14] Data transformation platform, https://www.datameer.com/.
[15] BI and analytics software, https://www.tableau.com/.
[16] Personal AI data assistant, https://datagpt.com/.
(Accessed: 24 January 2024).

Ultimately, it is worth mentioning that deploying the aforementioned products for a large user base can be pricey, as many of them calculate billing per user. Some tools offer free basic plans with limitations, like ⚒ Klipfolio[17] and ⚒ Zoho Analytics[18], or are heavily reliant on specific platforms, such as ⚒ Google Analytics[19].

In general, the proliferation of data preparation/wrangling tools in both industry and research have demonstrated the importance of data processing tasks and the success of the *low-code* concept. Martinez et al. (2023) affirmed the significant benefits of *low-code* in transforming data into information. However, these rewards come with inevitable costs (Section 2.1.1) and at the end, users still need programming knowledge to accomplish complicated features (Luo et al., 2021). In this case, accelerating programming tasks would be beneficial for both domain experts and developers. The next section discusses ML-based approaches for this purpose.

2.2 Advancing Programming with Machine Learning-based Approaches

Programming is a crucial phase in both the software development and data science life cycles. While *low-code* approaches enhance accessibility for a diverse range of users, from professional developers to domain experts (details in Section 2.1.1), modern IDEs facilitate the process while minimizing potential errors for developers. Essential features of an IDE include code editing, compiling, debugging, syntax highlighting (which uses different colors for keywords and relevant code tokens), linting to underline syntactic errors, and auto-completion based on built-in libraries (RedHat, 2019; Dehaerne et al., 2022).

However, many IDEs, without reliance on particular plugins, lack support for complicated tasks such as predicting subsequent code tokens from prior code or implementing functions from NL descriptions. To meet the rising demand for accelerating scripting to non-programming experts, various research directions have emerged, fueled by the rapid evolution of ML and technologies (Dehaerne et al., 2022; Yang et al., 2023; Sharma et al., 2024).

This section offers a concise overview on application domains of ML-based approaches for source code. It then summarizes the *naturalness hypothesis*, a key concept inspiring the advancement of ML models for code. Subsequently, popular traditional ML techniques and well-known deep learning methods for code-related tasks are outlined.

2.2.1 Application Fields

Numerous surveys have been conducted to exploit the abundance of studies in the domain of ML for source code (Allamanis et al., 2018; Le et al., 2020; Dehaerne et al., 2022; Yang et al., 2023; Sharma et al., 2024). Despite varied focuses and grouping methods, these surveys still share some overlapping application categories. In this dissertation, we mainly discuss research fields related to our work and briefly mention the remaining.

[17] Self-service BI, https://www.klipfolio.com/.
[18] BI platform, https://www.zoho.com/analytics/.
[19] Web analytics service, https://marketingplatform.google.com/about/analytics/. (Accessed: 24 January 2024).

> Among a plethora of application types in ML for source code domain, NL to code models and code completion approaches are our core interests.

Particularly, Dehaerne et al. (2022) classified ML-based code generation studies into three paradigms: description-to-code, code-to-description, and code-to-code, each encompassing one to multiple application types.

Description-to-code indicates approaches that synthesize code from non-code inputs, with NL to code techniques occupying the majority (Dehaerne et al., 2022; Wang et al., 2023b). Other categories involve producing scripts from input-output examples and converting images/sketched GUIs to code (Souza Baulé et al., 2020; Dehaerne et al., 2022).

Notable models and tools of the NL to code domain include TranX (Yin et al., 2018), CodeT5 (Wang et al., 2021c), GitHub Copilot[20], and Tabnine[21]. Listing 2.5 displays an example of translating NL comments to Python code using GitHub Copilot (v1.155.0) in Visual Studio Code (VSCode). The first two lines are requirements or description specified by users while the subsequent lines are code statements generated by Copilot.

```
1 # increase every element in the list of salaries by 1000
2 # and print the new list
3 salaries = [1000, 2000, 3000, 4000, 5000]
4 for i in range(len(salaries)):
5   salaries[i] += 1000
6 print(salaries)
```

produced by GitHub Copilot

Listing 2.5: Translating NL to Python code with GitHub Copilot (v1.155.0) in VSCode.

Code-to-description is also known as documentation generation or code summarization. These studies target to generate understandable NL description of a code snippet. Prominent models of this research direction comprise PLBART (Ahmad et al., 2021), CodeBERT (Feng et al., 2020), GitHub Copilot, and GPT-3.5 & 4[22], which power Copilot Chat[23].

Listing 2.6 represents some first lines of an explanation generated by GitHub Copilot (v1.155.0) and Copilot Chat (v0.11.1) in VSCode for the code snippet in Listing 2.5 (lines 3 – 6). The content is created by right clicking on the chosen code lines and selecting the options "`Copilot`" then "`Explain This`". For simplicity, Listing 2.6 displays solely a part of the explanation.

Code-to-code applications transform code from one form to another. Categories in this paradigm include automatically repairing buggy programs, translating across PLs, refactoring code to a more coherent format in the same PL, and suggesting next code tokens from previously typed tokens, i.e. code completion.

Specifically, multiple studies have investigated the code completion field and yielded positive results, such as PHOG (Bielik et al., 2016), Pointer Network (Vinyals et al., 2015), IntelliCode

[20] AI developer tool, https://github.com/features/copilot.
[21] AI coding assistant, https://www.tabnine.com/.
[22] OpenAI models, https://platform.openai.com/docs/models.
[23] Copilot update, https://github.blog/changelog/2023-11-30-github-copilot-november-30th-update/. (Accessed: 29 January 2024).

```
The active selection is a Python script that increases each value in the salaries list by 1000.
Here's a step-by-step breakdown:
* A list named salaries is defined with five integer values: 1000, 2000, 3000, 4000, and 5000. These values represent salaries.
* A for loop is initiated with i as the loop variable. The range() function [...]
```

Listing 2.6: Generating explanation for the code snippet in Listing 2.5 (lines 3 – 6) with GitHub Copilot (v1.155.0) and Copilot Chat (v0.11.1) in VSCode. Just a part of the text is presented for demonstration purpose.

Compose (Svyatkovskiy et al., 2020), Codex[24] (Chen et al., 2021), GitHub Copilot, and Tabnine. Listing 2.7 illustrates code completion using GitHub Copilot (v1.155.0) in VSCode. Namely, users implement a function for drawing a bar chart from provided data while Copilot predicts the code tokens on Line 6, from right after `plt.bar` til the end of the line.

```
1 import matplotlib.pyplot as plt
2 def generate_chart(xdata, ydata, xlabel, ylabel, labels, title):
3   plt.figure(figsize=(10, 5))
4   for i in range(len(ydata)):
5     # the cursor is placed right after "bar" when activating code completion feature
6     plt.bar(xdata, ydata[i], label=labels[i])
```

Listing 2.7: Completing Python code with GitHub Copilot (v1.155.0) in VSCode.

Other noteworthy application fields encompass code clone detection and vulnerability analysis (Sharma et al., 2024). The former seeks to identify duplicate code blocks and can be applied to various scenarios, such as detecting similar mobile applications or revealing license violations (Sajnani, 2016). Meanwhile, the latter focuses on identifying potential security issues in software products, which have severe impact on information safety (Zeng et al., 2020).

In this dissertation, we prioritize approaches that expedite programming for both domain experts and practitioners in data analysis and processing tasks. Whereas other application types are more developer-oriented, code completion and NL to code align best with our target users. Besides, cross-PL translation offers a promising direction to address the multi-PL roadblock for practitioners (details in Appendix A.3) and will be discussed as our future work (Chapter 8).

Despite distinct purposes, the mentioned applications share underlying techniques, ranging from traditional methods such as Decision Tree, k-Nearest Neighbor (KNN), Random Forests, and Reinforcement Learning, to advanced techniques like Deep Learning (DL) with Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), Convolutional Neural Network (CNN), Graph Neural Network (GNN), and Gated Recurrent Unit (GRU) (Dehaerne et al., 2022; Sharma et al., 2024). The following subsections outline notable techniques and models, primarily in the code completion and NL to code domains. But firstly, we summarize the *naturalness hypothesis*, a pivotal concept driving the advancement of ML for source code.

[24]The power behind GitHub Copilot.

2.2.2 The Naturalness Hypothesis

Hindle et al. (2016)[25] introduced the *naturalness hypothesis* for software:

> *Programming languages, in theory, are complex, flexible and powerful, but, "natural" programs, the ones that real people actually write, are mostly simple and rather repetitive; thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks.*

In other words, software can be considered as a form of human communication, making it possible to leverage techniques of Natural Language Processing (NLP) for code generation.

Hindle et al. (2016) utilized *n-gram* language models to illustrate their thesis. Typically, a language model assigns probabilities to code programs, considering source code sc as a sequence of code tokens $t_1 t_2 ... t_i ... t_n$, where each code token follows others with a specific probability. Therefore, the probability of the source code sc can be calculated based on the product of a series of conditional probabilities:

$$p(sc) = p(t_1)p(t_2|t_1)p(t_3|t_1t_2)...p(t_n|t_1...t_{n-1}) \tag{2.1}$$

Using *n-gram* models, the existence of token t_n is determined exclusively by a fixed-size window of $(n-1)$ preceding tokens. For instance, *3-gram* models estimate the likelihood of tokens $t_{i-2}t_{i-1}$ followed by t_i:

$$p(t_i|t_1...t_{i-1}) \simeq p(t_i|t_{i-2}t_{i-1}) \tag{2.2}$$

$$p(t_i|t_{i-2}t_{i-1}) = \frac{count(t_{i-2}t_{i-1}t_i)}{count(t_{i-2}t_{i-1}*)} \tag{2.3}$$

where $*$ is a wildcard, representing any tokens in the source code sc. Figure 2.3 demonstrates an example of obtaining token sequences from code statement `plt.figure(figsize=(10,5))` with *unigram* ($n = 1$), *bigram* ($n = 2$), and *trigram* ($n = 3$) models.

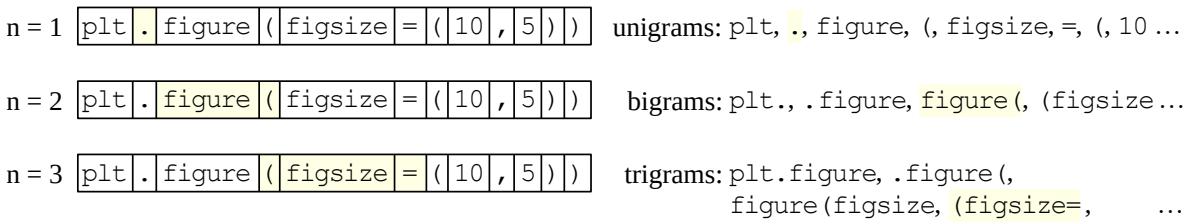


Figure 2.3: A simple example of generating code token sequences with *n-gram* models.

Despite the inherent data sparsity issue of *n-gram* models, which can be addressed by some smoothing techniques (Chen et al., 1999), Hindle et al. (2016) still could demonstrate that *n-gram* models can effectively capture the local regularities in code corpora and utilized this model to enhance the built-in suggestion engine of Eclipse IDE.

The *naturalness hypothesis* has yielded numerous ML studies for code with positive results such as Mikolov et al. (2013c), Bielik et al. (2016), Li et al. (2018), Svyatkovskiy et al. (2020),

[25] The original version of this paper was published in ICSE'12, noted by the authors.

Kim et al. (2021), and other methodologies presented in Chapters 4–6. Both conventional ML and DL methods discussed in the subsections below derive insights from this hypothesis.

2.2.3 Traditional Machine Learning Methods for Code Generation

Traditional ML, a subset of AI, employs various disciplines (e.g. statistics, probability theory) to tackle cognitive tasks like object detection or NL translation. Its techniques involve manual data feature extraction and hence are best suited for data with well-defined structures. Additionally, most of ML methods handle data with a shallow structure, characterized by at most one hidden layer (Janiesch et al., 2021; Wang et al., 2021a).

> Prominent classical ML approaches for code generation draw inspiration from probabilistic language models, aiming to capture both semantic and syntactic information in code.

Addressing the Lack of Semantic Representation

Despite the promising attempt made by Hindle et al. (2016) to analyze syntactic regularities in natural code, their study overlooked important semantic and structural information inherent in code statements, such as data types, token types, and relationships between tokens. The *n-gram* models used in their experiments focused solely on syntactic aspects, neglecting these critical semantic elements. Consequently, various methods have been explored to address this problem.

Parse trees with Context-Free Grammar. A notable method for encoding structural details involves parse trees generated by a Context-Free Grammar (CFG), a widely applied technique in NLP. The CFG technique is originally designed for modeling constituent structure (e.g. noun phrases) in natural languages (Jurafsky et al., 2008). We recall the formal definition of CFG here for future reference. A CFG, denoted as G, is defined by a tuple (N, Σ, R, S), as described in Table 2.2.

Table 2.2: The formal definition of Context-Free Grammar (Jurafsky et al., 2008).

Parameter	Meaning
N	a set of *non-terminal symbols* (or variables)
Σ	a set of *terminal symbols* (disjoint from N)
R	a set of rules or *productions*, each of the form $A \rightarrow \beta$, where A is a non-terminal symbol, β is a string of symbols from the infinite set of strings $(\Sigma \cup N)^*$
S	a designated *start symbol* and a member of N

Due to the mentioned regularities in software, CFG can be employed for code statements to generate the corresponding parse trees. For instance, Figure 2.4 displays a possible CFG for binary arithmetic expressions with four operations: addition, subtraction, multiplication, and division (Figure 2.4(a)), along with a parse tree for the expression $2*(4+5)$ (Figure 2.4(b)). This CFG includes two *productions* with E as the *starting symbol*, where E and Op are non-terminal symbols, and the rest are terminal symbols.

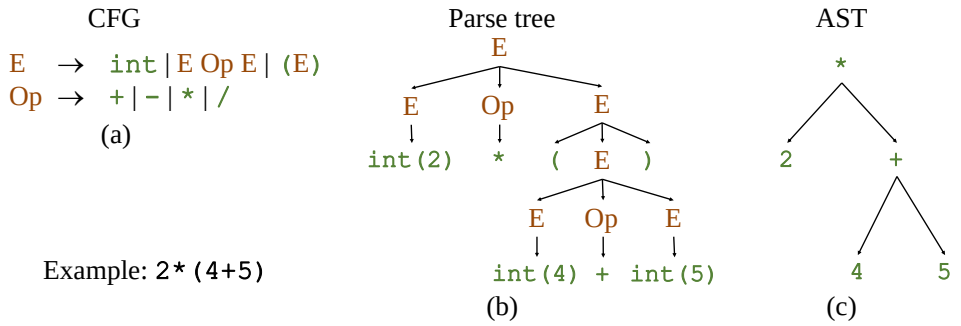


Figure 2.4: A CFG for binary expressions with four operations (a), alongside a parse tree (b), and an AST (c) of an example expression (Stanford, 2015; Pingali, 2023).

Semantic annotations. Nguyen et al. (2013) overcame the semantic representation issue by modeling regularities on semantic annotations (i.e. *sememes*) instead of textual representation of code token (i.e. *lexemes*). In their study, each token is annotated with its data type and semantic role. For instance, annotation of a code token named `str` with the role `variable` and data type `String` is `VAR[String]`.

Abstract Syntax Trees. Another popular form to convey structural information of code is Abstract Syntax Tree (AST), a condensed representation of a parse tree that is more suitable for later compiler stages due to the elimination of unnecessary intermediate nodes (Pingali, 2023). Figure 2.4(c) illustrates an AST for the expression $2 * (4 + 5)$ with the operators ($*$ and $+$) exhibited as internal nodes, in contrast to non-terminal symbol E in the parse tree of Figure 2.4(b). In practice, an AST is encoded in a programming language with node types, as shown in Figure 2.5(b).

Taking the advantage of the *naturalness hypothesis* of code, (Bielik et al., 2016) deployed the *n-gram* technique on ASTs and demonstrated that *n-gram* models with ASTs are more precise than *n-gram* modeling on the lexicalized representation of the program (i.e. the syntactic representation). Furthermore, AST serves as a widely adopted input form in various DL methods for code completion (details in Section 5.2).

By integrating semantic information into data representation, several probabilistic language models become applicable to code-related tasks. Two prominent models for code generation are outlined below, serving as the starting points for Chapter 4.

Probabilistic Context Free Grammars

While CFG excels in parsing, its effectiveness diminishes in assessing program correctness (Bielik et al., 2016). Probabilistic Context Free Grammars (PCFG) is an augmentation of CFG, slightly modifying the formal definition by assigning a probability p to each production $A \rightarrow \beta$:

$$A \rightarrow \beta \ \ [p] \tag{2.4}$$

where p is a number between 0 and 1, expressing the probability of generating β given the existence of non-terminal A, i.e. $p(\beta|A)$. Additionally, probabilities of all possible expansions of a non-terminal should sum up to one (Jurafsky et al., 2008).

$$p(\beta|A) = p(A \to \beta) = p(A \to \beta|A) \tag{2.5}$$

$$= \frac{count(A \to \beta)}{count(A)} \tag{2.6}$$

$$\sum_{\beta} p(A \to \beta) = 1 \tag{2.7}$$

PCFG finds application in various language modeling functions, such as speech recognition, machine translation, spelling correction, and augmentative communication (Jurafsky et al., 2008). Gvero et al. (2015) pioneered the use of PCFG in code synthesis, handling queries with mixed English and Java to generate Java code expressions. In addition, Bielik et al. (2016) applied PCFG for code completion in JavaScript. However, Bielik et al. also revealed that the combination of the *n-gram* technique and ASTs (mentioned above) outperforms PCFG in recommending both non-terminals and terminal AST nodes.

Probabilistic Higher Order Grammar

While *n-gram* models focus only on $(n-1)$ previous tokens, the production rules of PCFG are calculated solely based on the non-terminal nodes without incorporating information about the nodes themselves. Consequently, code tokens suggested by these techniques might be undesirable (Bielik et al., 2016). Probabilistic Higher Order Grammar (PHOG) overcomes these problems, generalizing PCFG by conditioning the production rules not only on a static parent non-terminal but also on a dynamically obtained context through navigation in an AST.

Formal definition. PHOG builds upon CFG and PCFG. Initially, the authors introduced Higher Order Grammar (HOG), a generalization of CFG that integrate a *context* γ into the production rules $A \to \beta$, creating a new rule:

$$A[\gamma] \to \beta \tag{2.8}$$

A HOG is defined by the tuple (N, Σ, R, S, C, f)[26], where N, Σ, R, and S are described as in Table 2.2. C is the contextual conditioning set where $\gamma \in C$, and f is the mapping function between all possible (partial) ASTs of a program and the corresponding conditioning set C.

Function f is implemented using programs in a DSL named TCOND, instructing traversal actions (e.g. up, down, left, right) while accumulating node-related information such as type, value, and position in an AST. The optimal f is determined by selecting the TCOND function with the minimum cost, considering factors such as log-probability from trained models and the number of instructions. For a detailed explanation of TCOND and its learning phase, we refer to the original paper of Bielik et al. (2016). Figure 2.5(b) illustrates an example of function f.

After determining the optimal function f, γ is the result from applying f to an AST starting at the current prediction point, forming a sequence of observations. Each observation in this sequence can be a non-terminal from N, a terminal from Σ, or a natural number from $\mathbb{N}$.

$$\gamma \in C = (N \cup \Sigma \cup \mathbb{N})^* \tag{2.9}$$

[26] We substitute f for p in the original paper to prevent confusion with the CFG definition.

Similar to PCFG, which introduces probabilities to each production rule of CFG, PHOG also assigns probabilities to each rule in HOG (Equation 2.10). Consequently, for a given context γ, the probabilities of available rules at a non-terminal sum to one.

$$A[\gamma] \rightarrow \beta \ \ [p] \tag{2.10}$$

$$p(A[\gamma] \rightarrow \beta) = \frac{count(A[\gamma] \rightarrow \beta)}{count(A[\gamma])} \tag{2.11}$$

$$\sum_{\beta} p(A[\gamma] \rightarrow \beta) = 1 \tag{2.12}$$

Illustrative example. Figure 2.5 presents a comparative example, highlighting the main distinction between PHOG and PCFG. The code snippet in Figure 2.5(a) is firstly parsed into the AST shown in Figure 2.5(b). To predict the next code token, depicted as `?` in Figures 2.5(a) and (b), PCFG and PHOG assess all possible production rules with the non-terminal `Property` in the left-hand side, as illustrated in Figure 2.5(c). Unlike PCFG, PHOG incorporates the context γ, determined by function f displayed under the AST in Figure 2.5(b).

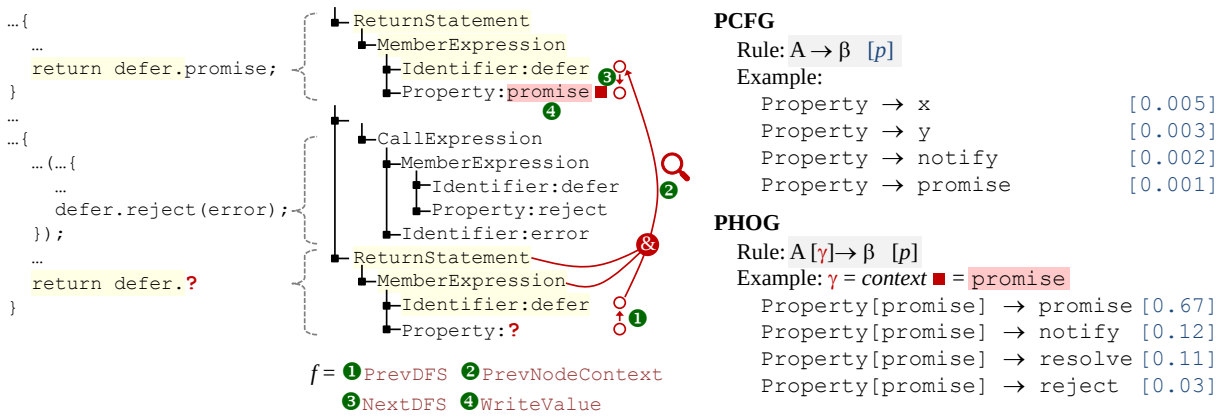


Figure 2.5: An example illustrating the difference in employing PHOG and PCFG for code completion, adapted from Figure 1 of Bielik et al. (2016).

In this example, the function f consists of four instructions: 1 `PrevDFS` shifts the current position to the previous node in Depth-First Search (DFS) traversal order, 2 `PrevNodeContext` identifies the previous node where the parent and grandparent share the same type and value, 3 `NextDFS` moves to the next node in the DFS order, and 4 `WriteValue` adds the value of the visited node to the accumulated context. The terminal node `promise` represents the context obtained by applying f to the AST.

Bielik et al. (2016) demonstrated superior performance of PHOG over both PCFG and *n-gram* on AST for predicting non-terminal and terminal AST nodes. Due to its effectiveness among traditional ML approaches, we adopted PHOG as a component in our proposed ensemble model for code completion, discussed in Chapter 4.

2.2.4 Deep Learning Techniques for Source Code

Deep Learning (DL), or Deep Neural Networks, is a subset of ML, targeting to mimic the learning process of the human brain through neural networks. In contrast to the conventional ML, DL can automatically discover feature representations from raw input data with minimal human effort, making it more adept at handling unstructured data. Furthermore, neural networks of DL models typically consist of more than one *hidden layer* (Janiesch et al., 2021).

> DL techniques for code generation leverage the power of Artificial Neural Networks and attention mechanism, embedding programs into vectors of numbers, with Transformer emerging as a game changer.

Artificial Neural Networks (ANNs) comprises processing elements called *neurons* or *nodes*, arranged in layers including an *input layer*, an *output layer*, and one or more *hidden layers* (Svozil et al., 1997). Numerous neural network-based models have been exploited for code-related tasks, such as Seq2seq (Sutskever et al., 2014), attention mechanism (Bahdanau et al., 2015; Luong et al., 2015), Pointer Network (Vinyals et al., 2015), Pointer Mixture Network (Li et al., 2018), and Transformer (Vaswani et al., 2017).

These models share the underlying neural network architectures, like Feed-forward Neural Network (FFNN), Recurrent Neural Network (RNN), and Long Short-Term Memory (LSTM). In short, FFNN is the most classical ANN with fully connected layers, transmitting information in one direction (forward) only (Abiodun et al., 2018). Meanwhile, RNN is a feed back ANN, designated for sequential data processing. Ultimately, LSTM, a type of RNN, aims to mitigate the *vanishing gradient problem* in RNN through gate units (Kelleher, 2019). A summary for these architectures and fundamental concepts of ANNs is provided in Appendices B.1–B.4.

Word embedding. Inputs of ANNs are vectors of numbers, requiring the conversion of text into number vectors when applying ANNs on source code. *Word embedding* technique (Mikolov et al., 2013b) is commonly employed for this purpose. Its central idea relies on the fact that words with similar surrounding words possess analogous meanings, which can be represented by the proximity of their vectors in space (further details in Appendix B.5).

Among various neural networks for NLP and source code, we present below an overview of models directly or semi-directly relevant to our work, particularly to our proposed methodologies in subsequent chapters.

Sequence-to-sequence (Seq2seq) Model

Seq2seq (Sutskever et al., 2014) is a widely adopted DL technique in NLP, particularly for machine translation (Kelleher, 2019). The model's architecture comprises two interconnected LSTM networks, known as encoder and decoder. The first LSTM processes an input sequence word-by-word to generate a vector representation, which is then utilized by the second LSTM to produce the output sequence. Figure 2.6 displays an abstraction of the encoder-decoder architecture with an illustrative example of translating a German sentence to English. In this depiction, enc_i and dec_j denote the i^{th} and j^{th} states of the encoder and decoder, respectively.

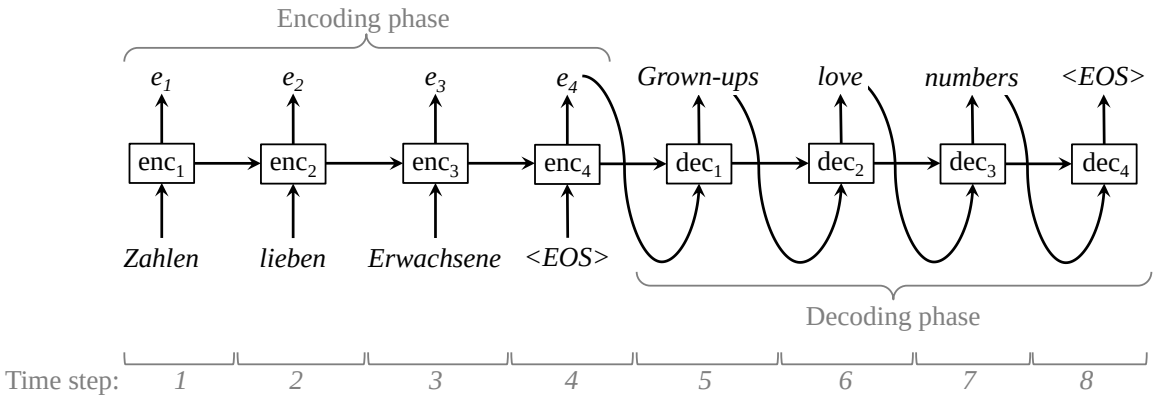


Figure 2.6: Abstract representation of the encoder-decoder architecture through time steps with an illustrative example.

Input words are handled by the model in reverse order to enhance performance and improve outcomes, as suggested by the authors. During the encoding phase, each word is processed by the encoder (i.e. the first LSTM) at every time step, generating a hidden state e_i, which is propagated forward to later time steps. The encoder completes processing upon encountering the end-of-sentence symbol, `<EOS>`. The resulting hidden state, i.e. vector e_4 in Figure 2.6, represents the input sentence.

In the subsequent phase, the vector representation e_4 serves as the initial input for the decoder (i.e. the second LSTM). The decoder is trained to sequentially generate each word of the output sentence. Each obtained word is subsequently fed back into the decoder as an input for the next time step. The decoding process terminates upon producing the `<EOS>` symbol.

Hidden state bottleneck. The encoder-decoder concept has formed the groundwork for numerous advanced language models (Yang et al., 2023). However, the original architecture faces a bottleneck as it must compress all information of an input sequence into a fixed-length vector (Bahdanau et al., 2015). This potentially hinders the model in coping with long sentences beyond the training corpus lengths. The next techniques were proposed to tackle this challenge.

Attention Mechanism

Bahdanau et al. (2015) introduced an extension for the encoder-decoder architecture called *attention mechanism*, which directs the model's focus to specific words at each time step rather than the entire input sentence. This enhances the model's memorization capacity and facilitates additional paths for back-propagation (Li et al., 2018). Figure 2.7 presents the high-level overview of the attention mechanism when predicting the output word out_t at time step t[27].

Distinctions compared to the Seq2seq model. The attention mechanism incorporates not only the last hidden state from the encoder but also previously generated hidden states. Particularly, given *seqlen* is the length of the input sequence, instead of forwarding to the decoder only the last hidden state e_{seqlen} (or e_4 in Figure 2.6), all the preceding encoder hidden states $e_1, e_2, e_3, ..., e_{seqlen-1}$ are also forwarded.

[27] For a step-by-step visual explanation of the attention mechanism, we prefer to a GitHub page of Alammar (2018b).

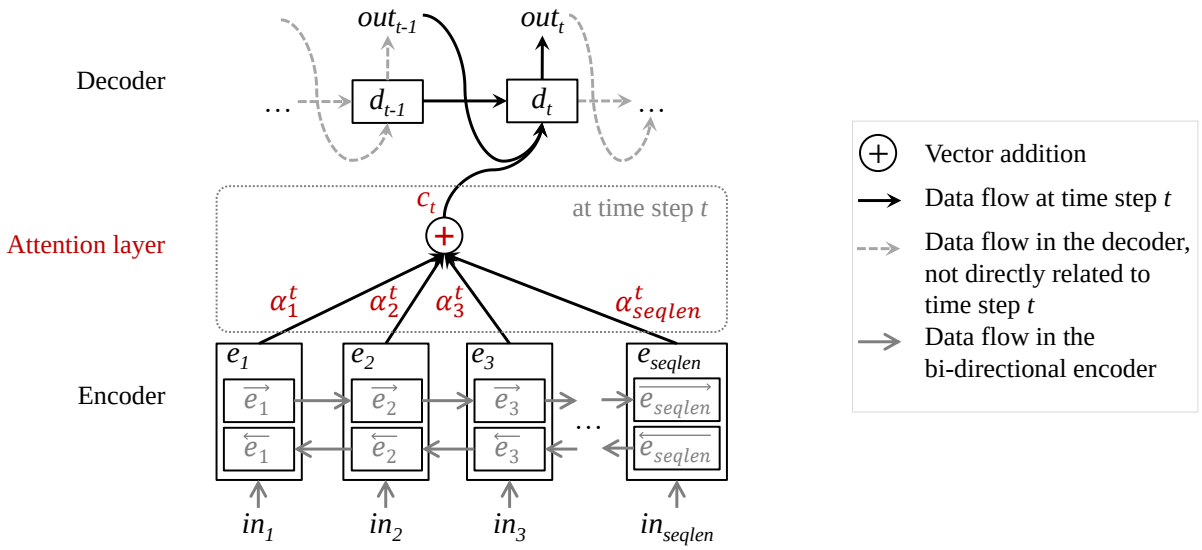


Figure 2.7: Abstract representation of the attention mechanism decoding at time step t, adapted from Figure 1 of Bahdanau et al. (2015) and Figure 4 of Weng (2018).

Additionally, Bahdanau et al. (2015) utilized a bidirectional RNN for the encoder, unlike the unidirectional approach in the classical Seq2seq model. This enables the encoder to capture information from both preceding and following words of the current word. The forward RNN processes the input sequence in order, while the backward RNN reads the sequence in reverse. Consequently, an encoder hidden state e_i is determined by concatenating the forward hidden state $\overrightarrow{e_i}$ and the backward hidden state $\overleftarrow{e_i}$ of the input word in_i, i.e. $e_i = [\overrightarrow{e_i}^\top; \overleftarrow{e_i}^\top]$.

Moreover, at time step t, the decoder with attention mechanism predicts the output word out_t using a context vector c_t, derived from all encoder hidden states e_i, the preceding decoder hidden state d_{t-1}, and the previously generated output word out_{t-1}. The decoder's current hidden state d_t is computed from d_{t-1}, out_{t-1}, and c_t according to Equation 2.13. The activation function f in this equation is employed by a gated hidden unit, similarly to LSTM units described in Appendix B.4. Formulas related to function f are omitted to streamline subsequent discussions.

$$d_t = f(d_{t-1}, out_{t-1}, c_t) \tag{2.13}$$

Key formulas. In general, the context c_t is a weighted sum of all encoder hidden states e_i:

$$c_t = \sum_{i=1}^{seqlen} \alpha_i^t e_i \tag{2.14}$$

$$\alpha_i^t = \text{softmax}_i(\text{score}(d_{t-1}, e_i)) = \frac{\exp(\text{score}(d_{t-1}, e_i))}{\sum_{k=1}^{seqlen} \exp(\text{score}(d_{t-1}, e_k))} \tag{2.15}$$

$$\text{score}(d_{t-1}, e_i) = A(d_{t-1}, e_i) = v_A^\top \tanh(W_A d_{t-1} + U_A e_i) \tag{2.16}$$

where the attention weight α^t at time step t is computed using a `softmax` function applied to a vector of attention scores, represented as $score(d_{t-1}, e_i)$ with $i \in [1, seqlen]$. Each score is obtained from an *alignment model* on d_{t-1} and e_i, i.e. $A(d_{t-1}, e_i)$, assessing the correspondence between the input around position i and the output at time step t. Bahdanau et al. (2015)

parameterized the alignment model A as a FFNN, jointly trained with all the other components of the system. In addition, $v_A \in \mathbb{R}^n, W_A \in \mathbb{R}^{n \times n}$, and $U_A \in \mathbb{R}^{n \times 2n}$ are trainable parameters, where n is the number of hidden units or the size of the hidden state (n = 1,000).

Variants of the attention mechanism. Various prominent methods have leveraged the proposed attention concept under different names, such as global attention (Luong et al., 2015), content-based attention in Pointer Network (Vinyals et al., 2015), context attention in Pointer Mixture Network (Li et al., 2018), and additive attention in Transformer (Vaswani et al., 2017). Besides, these methods slightly adjust attention scores and/or final output computation.

For instance, Luong et al. (2015) and Vinyals et al. (2015) simplified the vanilla attention mechanism by concatenating the context vector c_t and decoder hidden state d_t, in contrast to the deep output and maxout hidden layer approach by Bahdanau et al. (2015). Li et al. (2018) deployed attention as a pointer mechanism, while Vaswani et al. (2017) applied scaled dot-product attention rather than additive attention as proposed by Bahdanau et al. (2015). More distinctions between the vanilla attention mechanism and its variants proposed by Luong et al. (2015) are analyzed in Appendix B.6.

Pointer Network

Vinyals et al. (2015) introduced *Pointer Network*, utilizing the alignment model from Bahdanau et al. (2015) for content-based attention. This method addresses problems with discrete outputs corresponding to positions in the input sequence, such as finding planar convex hulls[28] and solving the traveling salesman problem[29].

Content-based attention. Their work was motivated by the constraint that output words in Seq2seq models, regardless of the presence of an attention mechanism, are determined by probabilities among words in a predefined dictionary (also called vocabulary). Consequently, training the model requires separate iterations for each dictionary size. This restricts the direct application of Seq2seq with attention to combinatorial problems, where the output dictionary size varies with the input sequence length.

Particularly, rather than integrating encoder hidden states into a context vector at each time step in the decoding phase, Vinyals et al. (2015) used attention as a pointer to select an element in the input sequence as the output, hence the name *pointer network*. Figure 2.8 shows an example of using Pointer Network to identify a planar convex hull.

Instead of generating a new output, the decoder chooses between input elements as the output via a pointer vector. Upon reaching the last item of the input sequence, a special symbol $\Rightarrow$ initiates the generation mode. The output of the current time step, in addition to hidden states, becomes input for the decoder at the subsequent time step. The process concludes upon encountering the symbol $\Leftarrow$.

[28] The *convex hull* of a point set is the minimal *convex polygon* covering all points. Traversing the boundary of the *convex polygon* in a clockwise direction always entails making left turns, `https://people.computing.clemson.edu/~goddard/texts/algor/A1.pdf`, (Accessed: 15 February, 2024).

[29] Determining the minimum-cost route for a salesman to visit all cities in a given list. Each city is visited only once (Hoffman et al., 2013).

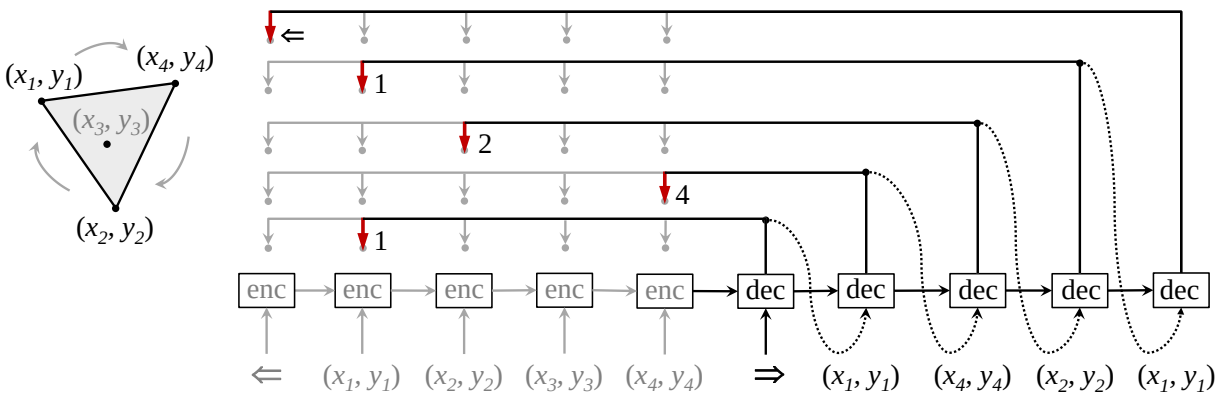


Figure 2.8: An example of using Pointer Network in plantar convex hull problem, adapted from Figure 1(b) of Vinyals et al. (2015).

Mathematical definition. In coordination with equations used by Bahdanau et al. (2015), the pointer mechanism proposed by Vinyals et al. (2015) can be formally defined as follows[30]:

$$u_i^t = \text{score}(d_t, e_i) = v_A^\top \tanh(W_A d_t + U_A e_i) \tag{2.17}$$

$$ptn^t = (u_1^t, u_2^t, ..., u_{seqlen}^t) \tag{2.18}$$

$$p(out_t | out_{<t}, inseq) = p(idx_t | idx_{<t}, inseq) = \text{softmax}(ptn^t) \tag{2.19}$$

where $v_A \in \mathbb{R}^n, W_A \in \mathbb{R}^{n \times n}$, and $U_A \in \mathbb{R}^{n \times n}$ are trainable parameters with n being the size of the hidden state ($n = 512$). idx_t signifies the index of the selected input element at time step t, with $idx_{<t}$ representing all previously referenced indices. $inseq$ denotes the input sequence, and $seqlen$ indicates the input length.

At time step t, the decoder estimates a score u_i^t for each encoder hidden states e_i, similarly to the mechanism of Bahdanau et al. (2015). Notably, Vinyals et al. (2015) utilized decoder hidden state d_t instead of d_{t-1} for score calculation (Equation 2.17). Subsequently, the pointer vector ptn^t is constructed from the set of all computed attention scores.

In this case, predicting an output is equivalent to determining the index of the most suitable element in the input sequence, given all previously chosen indices and the input sequence (Equation 2.19). Consequently, this objective is achieved by applying the `softmax` function to ptn^t, yielding a probability distribution over the input dictionary. In other words, the output dictionary depends on the length of the input sequence.

Pointer Mixture Network

Leveraging the power of LSTM, attention mechanism, and Pointer Network, Li et al. (2018) introduced a code completion model named *Pointer Mixture Network* for dynamically-typed PLs (e.g. Python and JavaScript). They aimed to tackle two main problems: (i) predicting next code token for dynamically-typed languages, and (ii) mitigating the *unknown word* or Out-of-Vocabulary (OOV) problem.

[30] We rename some parameters to make them consistent with preceding equations.

Unknown word issue. Neural language models normally generate output words based on a probability distribution across words in a predefined vocabulary (i.e. dictionary). However, computing this high-dimensional `softmax` can be computationally intensive. To mitigate this, the vocabulary typically includes only the K most frequent words in the corpus, with all others treated as OOV words (Li et al., 2018), represented by a special token (e.g. `<UNKNOWN>`). This issue is even more severe in source code due to the flexibility in naming identifiers by developers.

Global RNN and local pointer. Li et al. (2018) utilized an RNN with attention, named *global RNN component*, to generate code tokens from a sequence of prior tokens. To address the OOV problem, they deployed a Pointer Network, named *local pointer component*, to select an input item as the output word. This approach capitalizes on the local repetition patterns during programming of developers and the copy mechanism of the Pointer Network. Consequently, the model can predict the next code token by either generating it from the global vocabulary or pointing to an item in the input sequence (for OOV cases). Figure 2.9 illustrates the generating phase (i.e. decoding) of Pointer Mixture Network at time step t.

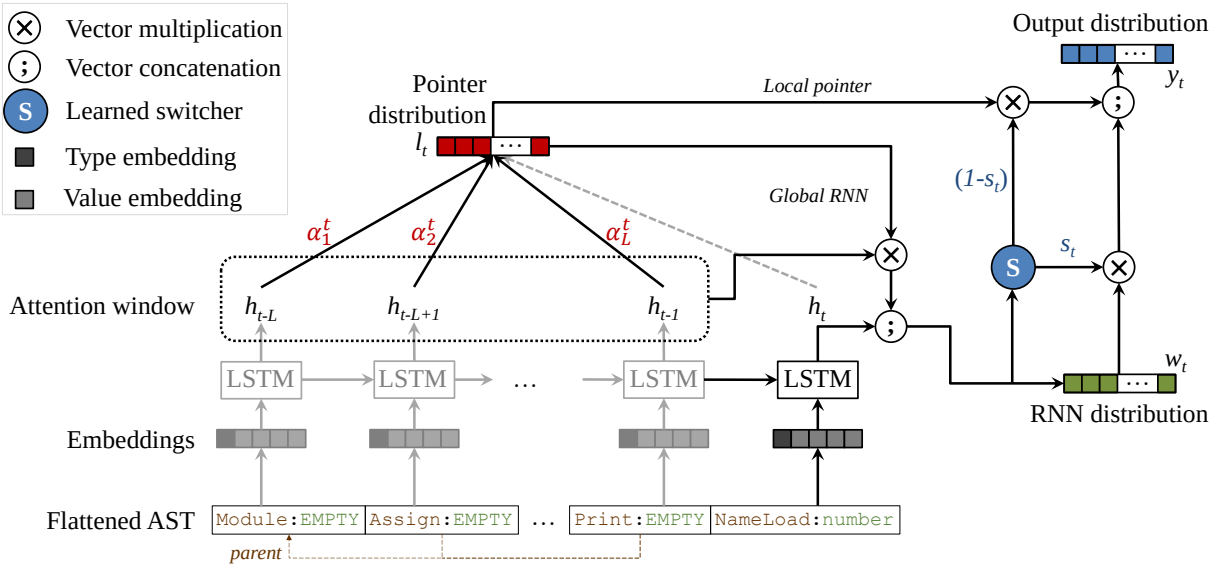


Figure 2.9: Predicting a next code token with Pointer Mixture Network at time step t, adapted from Figure 3 of Li et al. (2018).

Attention window. Instead of considering all previous tokens, the authors restricted the local context to an L-size range, named *context window* or *attention window* M_t. At time step t, the attention weight α_t and context vector c_t are calculated using preceding L hidden states $M_t = [h_{t-L}, h_{t-L+1}..., h_{t-1}]$, following the attention mechanism discussed above.

$$\text{score} = v^\top \tanh(W^m M_t + (W^h h_t)1_L^\top) \tag{2.20}$$

$$\alpha_t = \text{softmax}(\text{score}) \tag{2.21}$$

$$c_t = M_t \alpha_t^\top \tag{2.22}$$

where $v^\top \in \mathbb{R}^n$, $W^m \in \mathbb{R}^{n \times n}$, and $W^h \in \mathbb{R}^{n \times n}$ are trainable parameters with n denoting the size of the hidden state or dimension of h_t (n = 1,500). Besides, 1_L is an L-dimensional vector of ones.

It is worth noting that the three equations above are alternative representations of Equations 2.14 to 2.16, using matrix computations.

Source code as flattened ASTs. To capture structural information, each code snippet is converted into an AST, which is subsequently flattened to form a sequence of code tokens. Type and value of an AST node are separated by colon, as demonstrated in the bottom of Figure 2.9. Consequently, each prediction entails two tasks: predicting the next node type and suggesting the next node value.

Parent attention. To integrate the parent-child relationship among AST nodes (indicated by dashed lines beneath the flattened AST in Figure 2.9), the authors introduced a *parent attention* for code completion. At time step t, the model identifies the parent node within the attention window and retrieves its hidden state p_t, providing supplementary context information for subsequent output computation. The vector p_t is not displayed in Figure 2.9.

General workflow. Li et al. (2018) trained an embedding vector for each type and value of AST nodes before feeding them to the model. Utilizing the attention weight α_t (also serving as pointer distribution l_t), the global RNN and local pointer components derive their outputs accordingly. The former concatenates the current hidden state h_t and other context vectors, i.e. c_t and p_t, before computing the RNN distribution w_t (related formulas are omitted for brevity). Meanwhile, the latter operates similarly to the mechanism in Pointer Network.

The final result is a weighted concatenation of outputs generated by the two components above. At each prediction time step, a switcher S is learned based on the context information $(h_t; c_t)$, guiding component selection. Consequently, the output distribution y_t is computed as $y_t = [s_t w_t; (1 - s_t) l_t]$ with $s_t \in [0, 1]$. Given the promising results of Pointer Mixture Network, it was incorporated as a component in our code completion approach, detailed in Chapter 4.

Transformer - An Overview

Introduced by Vaswani et al. (2017), Transformer model revolutionized sequence transduction (i.e. sequence-to-sequence) models, attributed to its effectiveness and capacity for enabling parallelization during training, which is particularly beneficial for handling large corpora with lengthy input sequences. The Transformer model not only outperforms state-of-the-art models, whose architectures are based on complex recurrent or convolutional neural networks, but also significantly reduces the training time, attributed to its parallelizable nature.

Specifically, Transformer represents a pivotal shift by replacing recurrent structures with a multi-head self-attention mechanism to capture global dependencies between input and output. This model has catalyzed advancements in machine translation, code generation, NL understanding and other related disciplines (Yang et al., 2023). Notable derivatives of the Transformer architecture include BERT (Devlin et al., 2018), GPT (Radford et al., 2018), Codex (Chen et al., 2021), GitHub Copilot and ChatGPT.

Given its widespread adoption and relevance to our research, the subsequent section of this chapter focuses on elucidating the Transformer architecture and delineating noteworthy models derived from it for code-centric tasks.

2.3 Transformers and Beyond

The Transformer's architecture comprises encoder and decoder components, diverging from aforementioned approaches by exclusively relying on attention mechanisms (Vaswani et al., 2017). This design choice empowers the Transformer model to overcome critical challenges inherent in RNNs, such as long range dependencies, vanishing and exploding gradient problems (described in Appendix B.3), large number of training steps, and sequential computation due to RNN architecture (Poupart, 2019). Before outlining notable successors of the Transformer, we summarize key architectural aspects that facilitate its effectiveness and enable parallelization.

2.3.1 Vanilla Transformer

Vaswani et al. (2017) proposed the Transformer model as stacks of $N = 6$ identical self-attention and point-wise fully connected layers for both encoder and decoder. This approach mirrors the concept of stacking multiple RNN hidden layers on top of each other, where the output sequence of one layer feeds into the subsequent layer (Graves et al., 2013; Pascanu et al., 2013), enabling deep neural networks in space and resulting in more intricate representations.

Abstract-level Architecture

Figure 2.10(a) provides an overview of the Transformer's architecture.

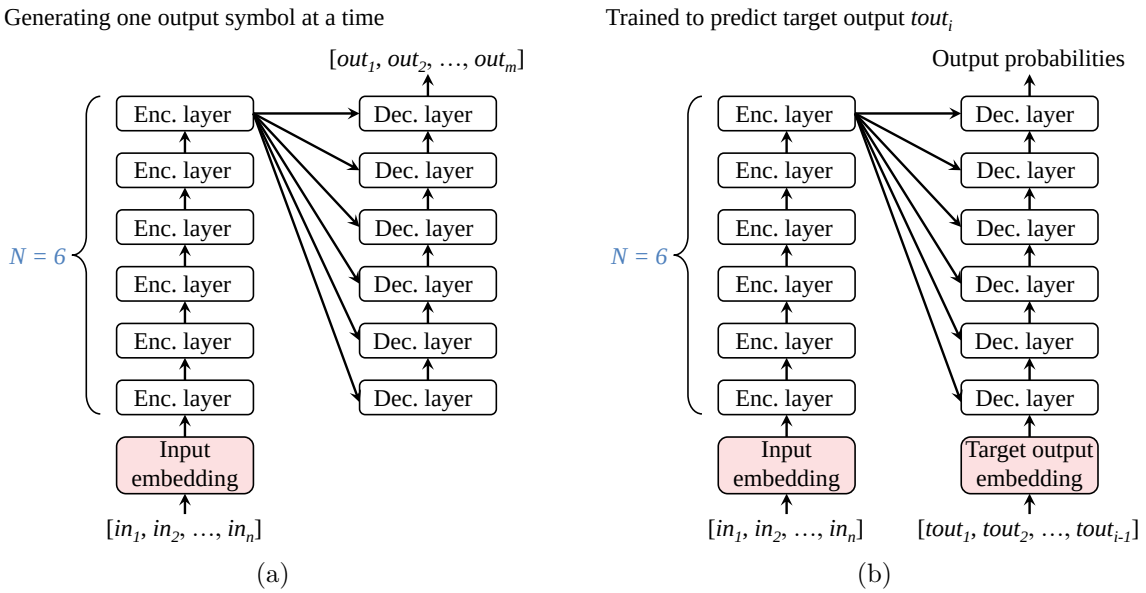


Figure 2.10: Transformer at an abstract-level view when generating an output sequence (a) and when being trained for predicting target output word $tout_i$ (b), adapted from Alammar (2018a) and Figure 1 of Vaswani et al. (2017).

Specifically, input sequences are converted into numerical vectors through learned embeddings before traversing the network. The input embedding links exclusively to the bottom encoder layer. Output of each encoder layer serves as input to the layer above it. Subsequently, output of the top encoder layer is distributed to all layers in the decoder stack, transmitting input sequence

information. Finally, the top decoder layer generates output symbols sequentially, similarly to the Seq2seq architecture (Alammar, 2018a).

To train the model, both input and target output sequences are embedded before being handled by the bottom layers of the encoder and decoder, respectively (Vaswani et al., 2017). Figure 2.10(b) illustrates the data flow while training the Transformer model to predict the next target output $tout_i$. Here, Transformer employs a technique known as *teacher forcing* to mimic the recurrent concept of RNNs, feeding the previous target output as input to the next state.

The underlying idea involves an assumption that at training phase, all preceding generated output words are correct and the model is trained to predict the next word based on this information. Therefore, to predict the next target output $tout_i$, previous target outputs (up to $tout_{i-1}$) are provided[31]. By deploying the teacher forcing technique, Transformer eliminates recurrences, facilitating parallelization in the training process and substantially accelerating training time (Poupart, 2019).

Encoder and Decoder Stacks

Figure 2.11 displays a closer look at the architectures inside each encoder and decoder layer.

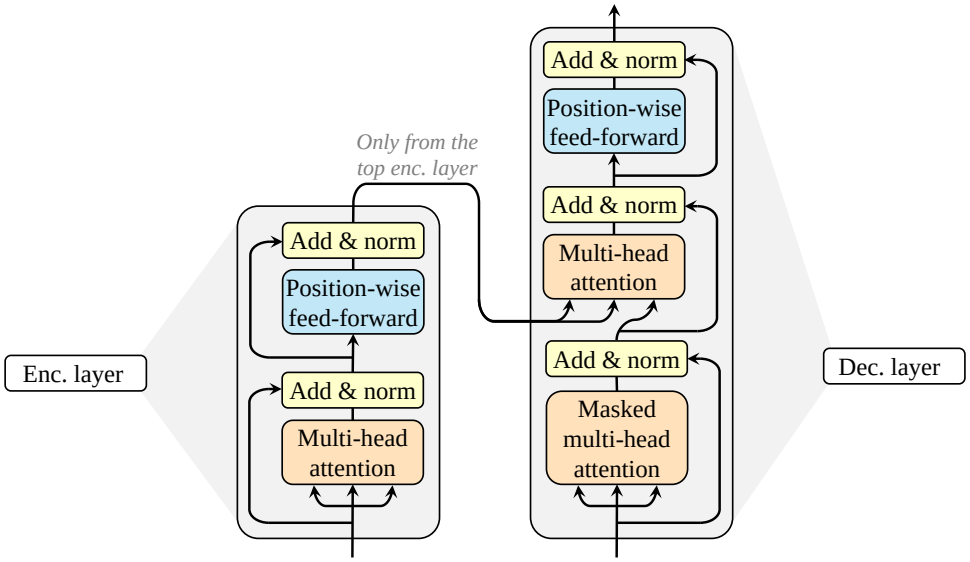


Figure 2.11: A deeper look at encoder and decoder layers of Transformer, adapted from Figure 1 of Vaswani et al. (2017).

Encoder. Each encoder layer encompasses two sub-layers: a multi-head self-attention and a position-wise fully connected FFNN, referred to as Multi-layer Perceptron (MLP) in other studies, such as Meng et al. (2022). Each sub-layer is encapsulated by a residual connection, followed by layer normalization to reduce the required number of training iterations (Poupart, 2019). Further details on these sub-layers are discussed below.

Decoder. In addition to the two sub-layers in each encoder layer, each decoder layer incorporates a third sub-layer, facilitating multi-head attention over the output of the encoder stack

[31]This explains the note "shifted right" in Figure 1 of Vaswani et al. (2017).

and acting as a reference to the input sequence during prediction (Vaswani et al., 2017). It is worth noting again that the connection from the encoder to the decoder stack occurs between the top (i.e. the last) encoder layer and each decoder layer.

Vaswani et al. (2017) also utilized residual connections and layer normalization for decoder sub-layers. Specifically, the bottom sub-layer is modified to a masked multi-head attention to omit the influence of subsequent symbols on the prediction. In order words, symbols following the prediction point are regarded as not yet generated and should be excluded during prediction.

Scaled Dot-product Attention

Before delving into multi-head attention, the significant sub-layer of encoder and decoder layers, we firstly summarize its core strategy, i.e. scaled dot-product attention.

Self-attention. Leveraging the concept of attention mechanism, the Transformer model employs *self-attention* to integrate positional relationships between symbols within a sequence into its representation (Vaswani et al., 2017). In essence, the vector representing a symbol in a sequence encapsulates information from other related symbols within the sequence.

To do this, each symbol in the sequence is associated with a *key-value* pair, with the current considered symbol serving as a *query*, where key, value, and query are all vectors. The objective is to identify all compatible keys with the query and retrieve their associated values, similarly to a database retrieval process (Poupart, 2019). The final output vector is computed as a weighted sum of the values, with the weight assigned to each value determined by the compatibility between the query and its corresponding key (Vaswani et al., 2017).

Mathematical expression. Given an input sequence $x = (x_1, x_2, ..., x_{seqlen}) \in \mathbb{R}^{seqlen \times d_x}$, adapting self-attention yields a new representation $z = (z_1, z_2, ..., z_{seqlen}) \in \mathbb{R}^{seqlen \times d_z}$, where d_x and d_z are the dimensions of x_i and z_i, respectively. Each element z_i is calculated using the following equations, according to Shaw et al. (2018) and Penke (2022):

$$z_i = \sum_{j=1}^{seqlen} \alpha_{ij}(x_j W^V) \tag{2.23}$$

$$\alpha_{ij} = \text{softmax}_j(\text{score}(x_i, x_j)) = \frac{\exp(\text{score}(x_i, x_j))}{\sum_{k=1}^{seqlen} \exp(\text{score}(x_i, x_k))} \tag{2.24}$$

$$\text{score}(x_i, x_j) = \frac{(x_i W^Q)(x_j W^K)^\top}{\sqrt{d_z}} \tag{2.25}$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d_x \times d_z}$ are parameter matrices. $x_i W^Q \in \mathbb{R}^{1 \times d_z}$ represents the query, while $x_j W^K$ and $x_j W^V \in \mathbb{R}^{1 \times d_z}$ with $j \in [1, .., seqlen]$ form the key-value pairs.

These equations resemble those employed in vanilla attention (Equations 2.14–2.16), with the distinction that Vaswani et al. (2017) utilized dot-product attention (Equation 2.25) instead of additive attention. This choice is justified by the faster and more space-efficient nature of dot-product attention. Notably, $\frac{1}{\sqrt{d_z}}$ acts as a scaling factor, ensuring that the dot product remains within a specific range, thereby preventing the `softmax` function from producing excessively small values. The name *scaled dot-product attention* is coined based on these described decisions.

Consequently, in practice, Equations 2.23–2.25 are executed via matrix multiplications to enhance efficiency, as follows:

$$z = \text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V \tag{2.26}$$

where queries, keys of dimension d_k, and values of dimension d_v are respectively packed into matrices $Q \in \mathbb{R}^{seqlen \times d_k}$, $K \in \mathbb{R}^{seqlen \times d_k}$, and $V \in \mathbb{R}^{seqlen \times d_v}$. Originally, d_k and d_v have the same value as d_z. However, to improve computational performance, Vaswani et al. (2017) introduced the concept of *multi-head attention*, significantly reducing the values of d_k and d_v (discussed below). In their experiments, $d_x = d_z = d_{model} = 512$.

Multi-head Attention

Figure 2.12 displays the multi-head attention architecture in the Transformer model. Rather than executing a single attention operation of dimension d_{model}, Vaswani et al. (2017) projected the queries Q, keys K, and values V linearly h times to dimensions d_k, d_k, and d_v respectively by using distinct learned linear projections.

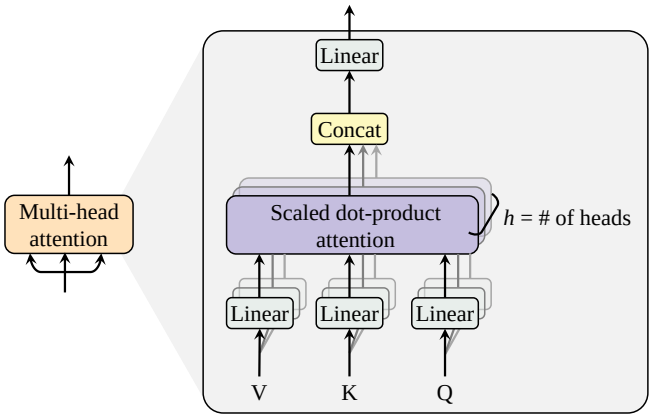


Figure 2.12: Architecture inside the multi-head attention of Transformer, adapted from Figure 2 of Vaswani et al. (2017).

Here, h is the number of heads and d_k as well as d_v are typically smaller than d_{model}. In simpler words, these linear layers simply change the space in which the queries, keys, or values reside. The objective is to iteratively apply the scaled dot-product attention with diverse parameter values, yielding multiple representation subspaces at various positions.

The projected queries, keys, and values are then concurrently employed to compute their scaled dot-product attention. This technique not only boosts the parallel processing capability of the model but also enables the generation of more complex representations (Poupart, 2019). Subsequently, the d_v-dimensional output values from all the subspaces are concatenated and linearly projected to create the d_{model}-dimensional output (top two layers in Figure 2.12).

Equation 2.26 for calculating the attention is adjusted to incorporate the multi-head concept as follows (Vaswani et al., 2017):

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \tag{2.27}$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.28}$$

where $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$ are trainable parameters. In their study, Vaswani et al. (2017) configured $h = 8$ parallel attention layers (i.e. 8 heads) and $d_k = d_v = d_{model}/h = 64$.

Applications of Multi-head Attention in the Transformer Model

The multi-head attention technique was leveraged in the vanilla Transformer model in three ways: encoder self-attention, decoder self-attention (masked), and encoder-decoder attention, as depicted in Figure 2.13.

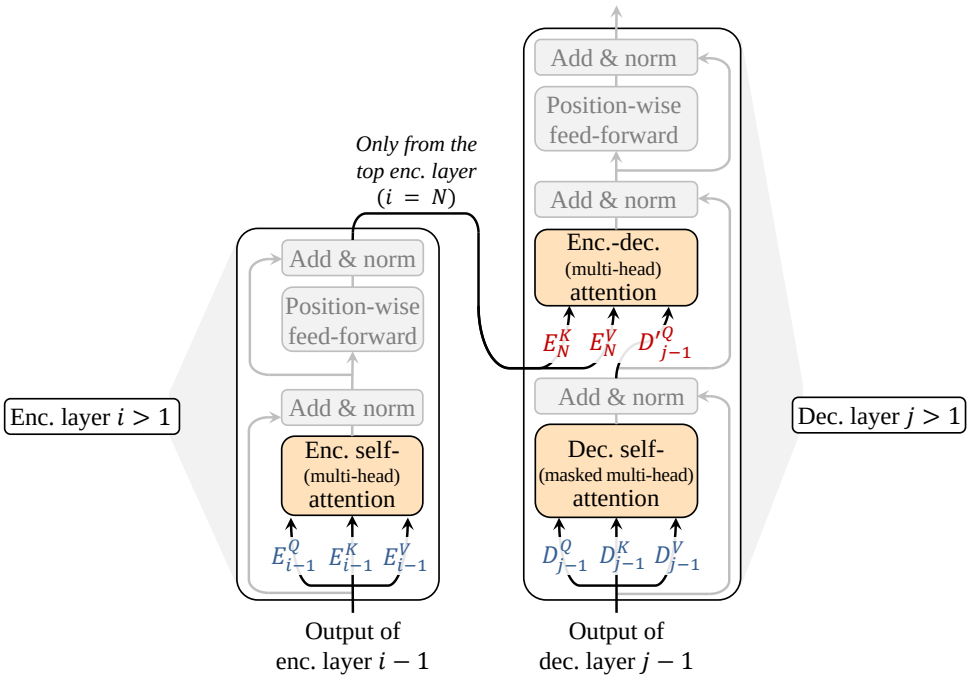


Figure 2.13: Utilizations of multi-head attention in Transformer architecture.

Encoder self-attention. The encoder self-attention sub-layers function similarly to the multi-head attention, with queries, keys, and values sourced from the previous encoder layer. These sub-layers empower each position in the encoder to attend to all positions in the preceding layer. In Figure 2.13, encoder layer i (with $i > 1$) receives the output of encoder layer $i - 1$ as input, including E_{i-1}^Q, E_{i-1}^K, and E_{i-1}^V.

Decoder self-attention. Meanwhile, each decoder layer's inputs encompass the previous decoder layer's output and the encoder's output. The masked self-attention sub-layers handle the former, preventing connections beyond the prediction point. These sub-layers allow each position in the decoder to attend solely to preceding positions and the current one. Equation 2.26 can be modified to form the masked attention as follows (Poupart, 2019):

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}(\frac{QK^\top + M}{\sqrt{d_k}})V \tag{2.29}$$

where $M \in \mathbb{R}^{seqlen \times seqlen}$ is a matrix of 0's and $-\infty$'s. Illegal connections are masked out by setting the values of relevant items in matrix M to $-\infty$. In Figure 2.13, the inputs to decoder layer $j > 1$ from preceding layer $j-1$ are denoted as D^Q_{j-1}, D^K_{j-1}, and D^V_{j-1}.

Encoder-decoder attention. Finally, keys and values from the encoder's output, together with queries from the prior decoder layer compose the inputs for the second attention unit in each decoder layer, so called encoder-decoder attention. Here, every position in the decoder can attend to all positions in the input sequence, analogous to the encoder-decoder attention mechanism in Seq2seq models (described in Section 2.2.4). In Figure 2.13, E^K_N, and E^V_N indicate the keys and values from the encoder (or the top encoder layer), while D'^Q_{j-1} represents the masked and normalized queries from the previous decoder layer.

Residual Connections and Layer Normalization

As mentioned above, Vaswani et al. (2017) stacked $N = 6$ identical layers for both encoder and decoder to increase network depth, which can cause *degradation* and gradient dependency issues during training. To address this, the authors implemented residual connections around each sub-layer (self-attention and position-wise feed-forward), followed by layer normalization to facilitate the training process. These residual connections and normalization sub-layers are displayed as "Add & norm" in Figures 2.11 and 2.13).

Residual learning. The residual connections in the Transformer model were inspired by the study of He et al. (2016), where the authors addressed the *degradation* problem in deep networks, using a deep residual learning framework. In particular, He et al. (2016) observed that as network depth increases, accuracy initially saturates but then declines rapidly. However, this degradation occurs independently of overfitting, with deeper models exhibiting higher training error upon further layer additions.

To tackle this obstacle, He et al. (2016) hypothesized that optimizing the residual mapping is easier than optimizing the original one. For instance, given $\mathcal{H}(x)$ as an underlying mapping to be fit by stacked layers, with x as the input to the first layer, He et al. (2016) explicitly let the stacked layers approximate a residual function $\mathcal{F}(x) = \mathcal{H}(x) - x$, instead of directly approximating $\mathcal{H}(x)$. The original function thus becomes $\mathcal{F}(x) + x$. In the context of the Transformer model, a sub-layer with a residual connection modifies the output as $x + \text{Sublayer(x)}$, where $\text{Sublayer}(x)$ designates the function performed by the sub-layer itself (Vaswani et al., 2017).

Layer normalization. Another challenge in training deep neural networks is "*covariate shift*", where gradients for a layer's weights depend heavily on the outputs of the previous layer, hindering convergence. To mitigate this dependency and speed-up the training process, Ba et al. (2016) introduced layer normalization, re-scaling the output of each layer to a uniform scale, hence stabilizing the hidden state dynamics. Particularly, for every vector $y^l = (y^l_1, y^l_2, ..., y^l_d)$ at layer l, the layer normalization is derived from mean and variance as follows (Ba et al., 2016; Xiong et al., 2020):

$$\text{LayerNorm}(y^l) = \frac{\text{g}^l}{\sigma^l}(y^l - \mu^l) + \text{b}^l \tag{2.30}$$

where g^l and b^l are gain and bias parameters of dimension d, while μ^l and σ^l signify the mean

and standard deviation of elements in vector y^l, formulated as follows:

$$\mu^l = \frac{1}{d}\sum_{k=1}^{d} y_k^l \tag{2.31}$$

$$\sigma^l = \sqrt{\frac{1}{d}\sum_{k=1}^{d}(y_k^l - \mu^l)^2} \tag{2.32}$$

Combining residual learning and layer normalization in the Transformer model, given vector a_i^l for $i \in [1..seqlen]$ and $l \in [1...N]$ as the d_{model}-dimensional output right after a self-attention sub-layer, the output after the "Add & norm" step of this sub-layer is determined as follows:

$$\bar{h}_i^l = \text{LayerNorm}(h_i^{l-1} + a_i^l) \tag{2.33}$$

where h_i^{l-1} is the output of the preceding layer $l-1$. The normalized vector $\bar{h}_i^l$ is computed using Equations 2.30–2.32, substituting y^l with $h_i^{l-1} + a_i^l$ and d with d_{model}.

Position-wise Feed-Forward Network or Multi-layer Perceptron

The second main type of sub-layers in each encoder or decoder Transformer layer is the position-wise FFNN, hereafter referred to as Multi-layer Perceptron (MLP). This layer is applied to each position separately and identically, consisting of two linear transformations with a Rectified Linear Unit (`ReLU`) activation in between (Vaswani et al., 2017).

Mathematical expression. Given $\bar{h}_i^l$ with $i \in [1..seqlen]$ and $l \in [1...N]$ from Equation 2.33 as the normalized output at position i and layer l of a self-attention sub-layer, the output m_i^l of the subsequent MLP sub-layer is obtained as follows:

$$m_i^l = \text{FFN}(\bar{h}_i^l) = \text{ReLU}(\bar{h}_i^l W_1^l + b_1^l)W_2^l + b_2^l \tag{2.34}$$

$$= \max(0, \bar{h}_i^l W_1^l + b_1^l)W_2^l + b_2^l \tag{2.35}$$

where $W_1^l \in \mathbb{R}^{d_{model} \times d_{ff}}$ and $W_2^l \in \mathbb{R}^{d_{ff} \times d_{model}}$ are the weight matrices; $b_1^l \in \mathbb{R}^{d_{ff}}$ and $b_2^l \in \mathbb{R}^{d_{model}}$ are the bias vectors. The input and output dimensionality is $d_{model} = 512$, while the MLP inner-layer has $d_{ff} = 2048$ hidden units. In addition, the linear transformations are consistent across positions but vary between layers, making this sub-layer equivalent to two convolutional layers with kernel size 1.

The significance of MLP sub-layers in the Transformer model. Notably, according to Geva et al. (2021), self-attention sub-layers constitute solely one-third of parameters in the Transformer model ($4d_{model}^2$ per layer), with the remainder allocated to MLP sub-layers ($8d_{model}^2$ per layer). This distribution highlights the significant role of MLP sub-layers in the architecture of the Transformer model.

Specifically, comparing Equations 2.26 and 2.34 without bias vectors reveals that FFN$(\cdot)$ and Attention$(\cdot)$ are structurally similar, differing primarily in their activation functions (`ReLU` vs. `softmax`). Geva et al. (2021) thus suggested that the input to FFN$(\cdot)$ can be interpreted as a query vector, with the two linear transformations of FFN$(\cdot)$ acting as keys and values.

In other words, the MLP sub-layers can be viewed as key-value memories of the Transformer model. Building upon this perspective, Dai et al. (2022) and Meng et al. (2022) elucidated the decisive role of these sub-layers in storing factual knowledge. The authors edited specific factual knowledge in pretrained Transformers, without any fine-tuning, by directly adjusting feed-forward weights while minimizing the impact on other knowledge. We refer to their original papers for the detailed studies.

Positional Encoding

To ensure that the Transformer model captures the input sequence order, positional encoding of each token is incorporated[32]. These encodings, with the same dimension d_{model} as the input embeddings, are added to the input embeddings before network processing. Namely, Vaswani et al. (2017) used sine and cosine functions to calculate the positional encoding for position pos at each element $k \in [0..d_{model}/2]$ of dimension d_{model} as follows:

$$PE_{(pos,2k)} = sin(pos/10000^{2k/d_{model}}) \tag{2.36}$$

$$PE_{(pos,2k+1)} = cos(pos/10000^{2k/d_{model}}) \tag{2.37}$$

There are various methods for positional encodings, such as learned positional encodings or concatenating positional and input embeddings instead of addition (Poupart, 2019). We refer to the original paper of Vaswani et al. (2017) for further details.

Ultimately, for comprehensive explanations of Transformer with illustrative examples, we refer to sources provided by Alammar (2018a), Doshi (2021), and Poupart (2019). The next subsection outlines notable models, derived from the Transformer architecture.

2.3.2 Successors of the Transformer Model

Transformer revolutionized sequence transduction models, setting a new state-of-the-art with its unique architecture. In particular, the parallelization capabilities are advantageous for large-scale corpus applications, aiding in the development of a category for DL models known as Large Language Models (LLMs)[33].

In recent years, Transformer derivatives have proliferated, divided into three groups: encoder-decoder, encoder-only, and decoder-only (Yang et al., 2023). The latter two types signify models that solely utilize either the encoder or decoder component of the Transformer model. Given our focus on code completion and NL to code translation tasks, we highlight prominent decoder-only models below and briefly touch upon encoder-only variants.

Decoder-only Models

GPT series. A year after the release of Transformer, OpenAI introduced Generative Pre-trained Transformer (GPT), a decoder-only model designed for language understanding tasks

[32] We omit this component in Figure 2.10 for simplicity.

[33] Large Language Models explained, https://www.nvidia.com/en-us/glossary/large-language-models/, (Accessed: 20 February, 2024).

(Radford et al., 2018). The model employs a semi-supervised approach, merging unsupervised pre-training with supervised fine-tuning.

During the training phase, the model is initially pre-trained on unlabeled data by predicting the subsequent word in a sequence based on preceding words (Yang et al., 2023). This process allows the model to learn initial parameter values for the neural network. These parameters are then fine-tuned for a target task using a corresponding supervised objective with labeled data.

Radford et al. (2018) showed that pre-training on a diverse corpus substantially enhances the model's acquisition of world knowledge, leading to improved performance on subsequent target tasks. Importantly, these target tasks do not need to align with the domain of the unlabeled corpus. Later versions of GPT have achieved remarkable results, being applied across diverse language understanding tasks, with later models aiming to surpass their predecessors in terms of parameter count and training set size (refer to OpenAI website for the details).

Advancements and challenges. Decoder-only models have rapidly advanced, dominating the evolution of LLMs with notable achievements (Yang et al., 2023). Outstanding models include GPT-2 to GPT-4, Codex (Chen et al., 2021), and ChatGPT from OpenAI, LaMDA from Google (Thoppilan et al., 2022), and LLaMA from Meta (Touvron et al., 2023a). ChatGPT gained significant attention upon its release by OpenAI[34], while LLaMA attracted interest by claiming the superiority of LLaMA-13B over GPT-3 (175B) across most benchmarks (Touvron et al., 2023a). Here, 13B and 175B are numbers of parameters in the neural networks.

However, the advent of GPT-3 has led to a preference among LLMs for closed-source models, posing challenges for academic researchers in conducting experiments (Yang et al., 2023). While Meta contributes notably to open-source LLMs, their LLaMA 2 model (Touvron et al., 2023b) supports a context window[35] of up to 4,096 tokens, half that of GPT-4 (8,192 tokens[36]), thereby limiting its capacity to lengthy sequences.

Furthermore, the constraint on prompt length remains a significant issue, particularly as prompts become longer (Jiang et al., 2023b). Ultimately, the demand for explainability in LLMs persists as a prominent yet challenging issue (Bird et al., 2022; Anaconda, 2023).

Encoder-only Models

BERT model. Exploiting the advantage of semi-supervised approach, Devlin et al. (2018) proposed a language representation model, called Bidirectional Encoder Representations from Transformers (BERT). Unlike GPT, BERT adopts only the encoder side of Transformer, aiming to demonstrate that pre-trained representations diminish the necessity for task-specific architectures. Fine-tuning with an additional output layer enables BERT to achieve state-of-the-art performance across various tasks such as answering questions and language inference.

Specifically, BERT employs two pre-training tasks. The first one involves utilizing Masked Language Model (MLM) to predict randomly masked words in a sentence based on surrounding

[34] A record of ChatGPT, https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/.
[35] Number of tokens an LLM can consider at a time.
[36] GPT-4 and GPT-4 Turbo, https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo. (Accessed: 20 February, 2024).

context, producing bidirectional representations. This differs from GPT's left-to-right approach. The second task is next sentence prediction, developed to capture the relationship between two sentences (Devlin et al., 2018).

Fading dominance. Several models, fueled by BERT, have emerged for code-centric tasks, including code completion with RoBERTa (Ciniselli et al., 2021b), code clone detection (Sharma et al., 2022), and NL code search or code documentation generation with CodeBERT (Feng et al., 2020). However, following the rapid proliferation of encoder-only models catalyzed by BERT, this domain has progressively receded, outclassed by decoder-only models (Yang et al., 2023).

Given the substantial impact of Transformer on the domain of ML for source code, our studies presented in Chapters 5 and 6 focus on enhancing specific aspects of Transformer-based models. The subsequent section summarizes and concludes this chapter.

2.4 Summary

This chapter provides background knowledge and related work, forming a foundation for our proposed methodologies in subsequent chapters. Key highlights include:

***Low-code* and *no-code* techniques** exhibit their effectiveness in expanding opportunities for domain experts to accomplish their data analysis and processing tasks with minimal to no programming effort. Besides certain benefits, these methods come with inherent limitations, such as *vendor lock-in*, restricted access to source code for cross-platform sharing, and limited customization. The practical side of our work aims to mitigate these issues, targeting both researchers and practitioners.

***External* DSLs**. Despite requiring some scripting knowledge, DSLs, the backbone of *low-code* and *no-code* techniques, demonstrate the superiority in conveying business logic and ease of use, compared to GPLs. While *internal* DSLs, i.e. a special way of using GPLs, require relevant knowledge of the implementing language, *external* DSLs are more suitable for domain experts, attributed to the separation of DSL scripts and underlying GPLs (i.e. host languages). Accordingly, *external* DSLs align with our objectives and serve as the groundwork for our deployed tool in Chapter 3.

ML-based approaches. In terms of research contributions, ML-based methods have emerged as the right direction for our work with numerous models exploited for code-related tasks. Particularly, methods for code completion and NL to code translation are our primary concerns. These approaches share the foundational concept known as the *naturalness hypothesis*, which leverages the repetitiveness in GPLs to apply NLP techniques to source code.

Tailored ML models. Before the widespread adoption of GPT, PHOG and Pointer Mixture Network excelled beyond traditional ML and DL models, respectively. Consequently, these two models are integrated into our proposed model in Chapter 4. Our later studies are influenced by Transformer and its successor, GPT. Therefore, this chapter features a specialized section dedicated to analyzing the Transformer architecture, setting the stage for comprehending our work in Chapters 5 and 6.

The next four chapters delve into our contributions in this dissertation.

Part II

Practical Contributions

NLDSL Extension

Chapter 3

Accelerating Programming for Data Analysis Tasks with Low-code Approaches in Practice

Domain-Specific Languages (DSLs), or highly abstracted programming languages tailored for specific application domains, are key foundations of *low-code* development. These languages demonstrate their success in facilitating collaboration between domain experts and developers in various fields, including data science. However, developing and maintaining DSLs requires substantial effort and expertise in both domain knowledge and language development.

In this chapter, we present *Natural Language to Domain Specific Language (NLDSL)*, a Visual Studio Code (VSCode) extension aimed at addressing the above challenges in constrained DSLs that model pipelines of operations. Building upon a tool of our research group, introduced by Andrzejak et al. (2019a), this extension is freely accessible on the Visual Studio Marketplace, representing our practical contributions to this dissertation.

Section 3.1 reiterates our motivation, while Section 3.2 summarizes the design of the vanilla *NLDSL* tool. The enhancements introduced in the *NLDSL* extension, in contrast to the base version, are detailed in Section 3.3. Subsequently, Section 3.4 outlines key aspects of disseminating the VSCode extension. In addition, preliminary evaluation, potential enhancements, and our response to the first core research question (CRQ1) are discussed in Section 3.5. Ultimately, we conclude the chapter in Section 3.6.

3.1 Introduction

***Low-code* techniques** are widely-adopted across domains, attributed to their ability to simplify scripting tasks for domain specialists with limited programming proficiency (Luo et al., 2021; Johannessen et al., 2021; Di Ruscio et al., 2022; Hirzel, 2023). However, users of *low-code* tool-kits usually encounter problems including lack of customization and dependency on specific vendor's environment (so called *vendor lock-in*), leading to potential difficulties in transitioning between platforms (Luo et al., 2021; Elshan et al., 2023; Martinez et al., 2023). Furthermore, certain *low-code* tool-kits pose challenges for developers in terms of maintenance and debugging (refer to Section 2.1.1 of Chapter 2 for further details).

Facilitating DSL development. The backbone of *low-code* techniques lies in DSLs. These highly abstracted GPLs designed for specific domains not only boost development efficiency and facilitate software artifact reuse but also act as shared languages among domain experts and developers, alleviating communication hurdles between these user groups (Fowler, 2010; Mernik

et al., 2005; Andrzejak et al., 2019a). However, DSL development is challenging and demands significant effort, expertise, and comprehensive consideration of factors ranging from business to technical aspects (Mernik et al., 2005; Alves, 2023). For more details on DSLs and their attributes, we refer to Section 2.1.2 of Chapter 2.

To mitigate the aforementioned issues of *low-code* and DSLs while targeting both domain experts and developers, facilitating development for DSLs is a promising approach. Hirzel (2023) highlighted that exposing DSLs for *low-code* allows users to test, audit, and share them across applications, thereby overcoming *vendor lock-in* and facilitating platform transitions. Moreover, enabling users to customize and integrate tailored DSLs in a *low-code* fashion can lessen the issue of limited customization in *low-code* tool-kits.

Python-based NLDSL, a tool developed by Andrzejak et al. (2019a) in our research group, demonstrates positive outcomes for the above approach. Besides, software solutions employing pipelines of operations are common in modern data processing and analysis libraries, indicating a broad spectrum of potential applications for pipeline-oriented DSLs. The proposed structure of DSLs in vanilla *NLDSL* leverages this concept, where each statement encompasses a chain or pipeline of operations.

In particular, the syntax description and implementation of DSL operations in this tool are integrated into annotated and concise Python functions, streamlining extension and maintenance for developers. To support domain experts in customizing their DSLs, vanilla *NLDSL* provides a mechanism to define statements at DSL level, serving as *first-class*[1] DSL entities.

Additionally, vanilla *NLDSL* supports code completion for DSL tokens, based on predefined syntaxes, within IDEs that implement Microsoft's Language Server Protocol (LSP). Moreover, DSL statements are embedded into GPLs as comments and translated to such GPLs during editing. Listing 3.1 presents an example of DSL designed for Pandas, where the first line contains the DSL statement, prefixed with `##`, and the second line displays the generated Python code.

```
1 ## on data | select columns 'Country/Region', 'Confirmed' | head 10 | show
2 print(data[['Country/Region', 'Confirmed']].head(10))
```

Listing 3.1: An example of a DSL statement for Pandas.

Selecting columns `Country/Region` and `Confirmed` of the dataframe `data` and printing the first ten rows.

Contributions of this work. We leveraged the promising results of the vanilla *NLDSL* tool and released it as a free VSCode extension in 2020, receiving encouraging responses from the community. Furthermore, we have augmented the tool with advanced features, additional Deep Learning (DL)-related DSLs, and a wizard for user-defined DSL development. This extension is being continually expanded with contributions from our practicum and Bachelor students. Prominent advancement of the published extension, compared to the vanilla *NLDSL*, include:

- *A wizard for DSL development.* We created a DSL development wizard that enables users to design their own DSLs using an Excel template and import these DSLs into the

[1] First-class values in a programming language are those that can be passed as parameters, returned from functions, or assigned to variables (Scott, 2000).

extension. Users can conveniently share their DSLs by sharing the utilized Excel file. This feature represents a significant addition compared to the original *NLDSL*.

- *Deep learning DSLs.* The initial release of *NLDSL* (v0.1.0) included DSLs for Pandas and PySpark, encompassing key processing steps from widely-used tutorials. The latest version (v0.5.0) of *NLDSL* extends its capabilities to incorporate alpha-stage DSLs for PyTorch and TensorFlow. These new DSLs cover fundamental DL tasks, including model creation, training, and evaluation.

- *Additional features for code completion.* In addition to the standard code recommendation based on predefined DSL syntaxes in vanilla *NLDSL*, we have introduced supplementary features: type provider and path completion for Comma-separated Values (CSV) files, library initialization via declaration of *target code* (e.g. Pandas or PySpark), and DSL grammar recall during editing. Details on these features are provided in Section 3.3.2.

- *DSL grammar adjustment.* We adjusted some DSL grammars to handle complex test cases, including the independent support for `group by` and `apply` operations in Pandas and PySpark DSLs. This modification allows users to specify arguments for the `apply` operation, a departure from the original version.

- *Common handling.* To facilitate developers in maintaining the extension, we restructured the underlying workflow. Each additional feature is now managed by a handler, with its settings centralized in a common location.

- *Syntax highlighting.* To improve DSL statement readability, we color-coded parameters and predefined functions within DSL statements instead of maintaining a uniform comment color, as shown in Listing 3.1. Additionally, we introduced a synchronization indicator to highlight discrepancies between a DSL statement and its generated code line. This feature is newly implemented in *NLDSL* compared to its initial version.

Notably, in addition to overcoming the *programming barrier* </> through DSL utilization, *NLDSL* enables users to seamlessly switch between sequential and parallel data processing modes, by adapting a unified structure for Pandas and PySpark DSLs, thereby mitigating the *scalability problem*. Similarly, DSLs for PyTorch and TensorFlow share a common grammar, facilitating smooth platform transitions for users without necessitating re-coding, alleviating the *reuse problem* (as defined in Chapter 1, Section 1.1). We outline key aspects in the design of the vanilla *NLDSL* in Section 3.2 before exploring the deployed extension in subsequent sections.

3.2 Python-based (vanilla) NLDSL Tool

This section provides an overview of the architecture and supported DSL structure of the vanilla *NLDSL*, along with its core features. This serves as a foundation for comprehending the advanced features introduced in the *NLDSL* VSCode extension. Further implementation details of the vanilla *NLDSL* can be found in the original paper of Andrzejak et al. (2019a).

3.2.1 Tool Architecture

The *NLDSL* tool comprises two primary components: (i) *NLDSL.lib*, a library accelerating development of pipeline-oriented DSLs, and (ii) *NLDSL.edit*, an environment enabling DSL editing and in-IDE code generation. Figure 3.1 depicts the architecture of the *NLDSL* tool.

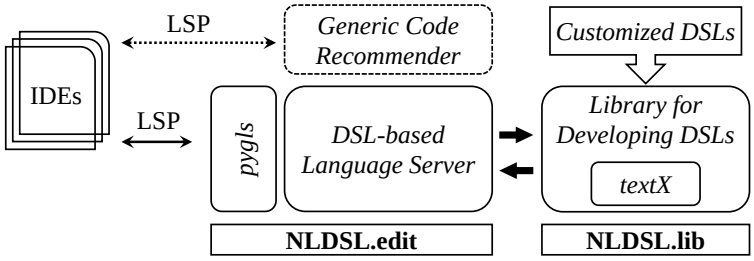


Figure 3.1: Architecture of *NLDSL* tool, adapted from Figure 1 of Andrzejak et al. (2019a).

The Python-based library *NLDSL.lib* utilizes the *language workbench*[2] textX (Dejanović et al., 2017) for low-level DSL parsing. *NLDSL.lib* can also operate independently of *NLDSL.edit*, generating Python code from customized DSL files through an integrated compiler. Section 3.2.3 explains in detail the customization of DSLs as inputs for *NLDSL.lib*.

Meanwhile, *NLDSL.edit* employs *NLDSL.lib* and pygls[3], a pythonic generic implementation of the LSP, to offer code completion for LSP-supported IDEs. Upon receiving a request of code completion from a DSL line (starting with `##`), the DSL-based Language Server is activated accordingly. Otherwise, if the request originates from a code line, it is directly forwarded to a generic code recommender (indicated by the dashed rectangle in Figure 3.1).

3.2.2 DSL Structure

As reviewed in Section 3.1, the vanilla *NLDSL* supports *external* DSLs, modeling pipelines of operations. This is attributed to the advantages of *external* DSLs over *internal* DSLs in exempting users from direct interaction with GPLs, simplifying scripting tasks (further details in Section 2.1.2 of Chapter 2). Additionally, supported DSLs are incorporated into GPLs as comments, hence the term *embedded external* DSLs (as mentioned in Chapter 1, Section 1.4).

Particularly, *NLDSL* assists two types of *external* DSLs: *evaluation* and *definition* DSL statements. The former denotes statements translatable into executable code (e.g. Python), while the latter provides an ad-hoc method to extend DSLs using internal functions. Figure 3.2 illustrates the grammar for the *evaluation* statements.

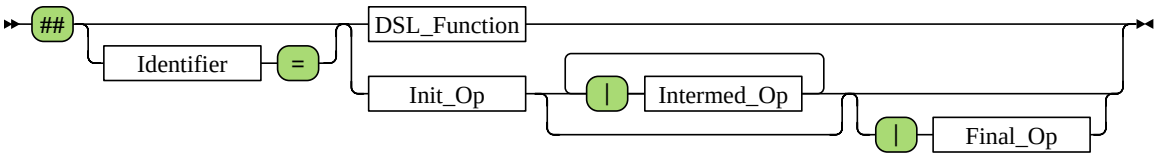


Figure 3.2: Grammar of *evaluation* DSL, adapted from Figure 2 of Andrzejak et al. (2019a).

[2] *Language workbenches* facilitate the creation and modification of PLs, as discussed further in Section 2.1.2.
[3] A generic Language Server framework, `https://pypi.org/project/pygls/`, (Accessed: 26 February 2024).

***Evaluation* statements**. The statement begins with `##`, indicating a comment line in Python (for other languages such as Java, it would be double forward slashes `//`). Subsequently, an optional assignment (`Identifier =`) is followed by an expansion involving two cases. The first case entails a function call to an internal DSL function (`DSL_Function`), specified by a *definition* statement. The second case comprises a pipeline of operations, delimited by the pipe symbol (`|`), including a mandatory initialization operation (`Init_Op`), an arbitrary number of intermediate operations (`Intermed_Op`), and an optional final operation (`Final_Op`).

Each operation type can be defined and implemented by a specific DSL. The initialization operation designates an object or data for processing. For illustrative purposes, Andrzejak et al. (2019a) utilized `"on <Identifier>"` as an `Init_Op` to indicate the input object or variable for the pipeline. The first line of Listing 3.1 provides an example of an *evaluation* DSL statement.

***Definition* statements** empower end-users to extend DSLs by creating new DSL functions using existing operations, resembling parameterized functions at the DSL level. A definition encompasses a declaration followed by the equal symbol (`=`) and a sequence of existing DSL operations or functions on the right-hand side. Listing 3.2 outlines the grammar for the left-hand side of the *definition* statements using Extended Backus-Naur Form (EBNF).

```
1 lhs      ::= "#$" name (keyword | var | expr | varlist)*;
2 varlist  ::= "$" identifier* "[" (keyword | var)+ "]";
3 var      ::= "$" identifier;
4 expr     ::= "!" identifier;
```

Listing 3.2: Grammar for the left-hand side of *definition* DSL statements in EBNF, adapted from Andrzejak et al. (2019a).

A declaration statement starts with `#$`, followed by the function name and zero to multiple expansions. Each expansion can be a DSL keyword, variable, expression, or variable list, as detailed in Listing 3.2. An example for this type of DSL statements with its usage scenario is presented in Section 3.2.3, alongside core features of the vanilla *NLDSL* tool.

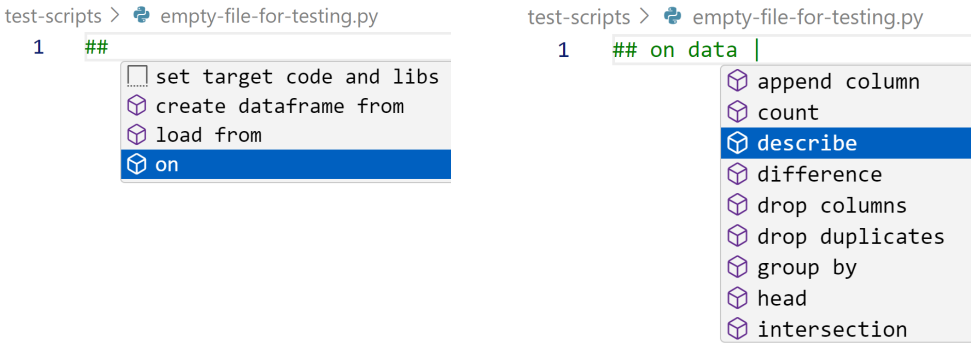
3.2.3 Core Functionalities

Andrzejak et al. (2019a) proposed the initial version of the *NLDSL* tool with three key features: (i) DSL code completion (at operation and statement levels), (ii) data inspection and preprocessing with Pandas and PySpark DSLs, and (iii) customization of DSLs through expression rules for developers and via internal functions for end-users.

DSL Code Completion

To enable the code completion feature for DSL statements, the key combination `Ctrl + space` should be pressed on a DSL line (starting with `##`). The vanilla *NLDSL* supports two types of code completion: at the DSL operation and DSL statement levels. The former suggests appropriate DSL operations based on the typed tokens, while the latter translates valid DSL statements into executable code.

Operation completion. As typical modern IDEs, the DSL-based Language Server (Figure 3.1) generates a list of potential DSL operations, accompanied by documentation, examples, and grammar definition for each operation upon activation. After selecting an operation, extension is achieved by appending the pipe symbol (|) and repeating the process. Figure 3.3 illustrates the recommendation lists provided by the DSL-based Language Server. The returned list is dynamically adjusted based on the entered operations.



(a) (b)

Figure 3.3: Code completion at operation level with *NLDSL* takes place at the beginning of the pipeline (a), and after an initialization operation (b).

Statement completion. Once a valid DSL statement is formed, it can be translated into executable code by selecting the option marked with the symbol ⬚ upfront. A DSL statement is considered valid when it conforms to a predefined grammar. Alternatively, the statements can be expanded further by choosing other options. Figure 3.4 depicts a Python code generated for a DSL statement describing a Pandas dataframe named `data`.

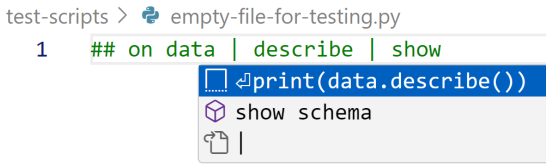


Figure 3.4: Code completion at statement level with *NLDSL*.

DSLs for Pandas and PySpark

Pandas and PySpark (built on Apache Spark) are well-known libraries for data analysis. While PySpark (or Spark in general) excels with large-scale datasets owing to its distributed computing ability, Pandas is preferable for smaller, tabular datasets (Zaman, 2023). By offering a unified set of DSL functions for both libraries, Andrzejak et al. (2019a) allow users to seamlessly transition between sequential and parallel processing scripts without needing to understand the underlying mechanisms, thereby mitigating the *scalability problem* (defined in Chapter 1, Section 1.1).

Specifying target code. To begin using the designed DSL for data manipulation, users must designate the *target code*, which indicates the code to which DSL statements will be

translated. The vanilla *NLDSL* offers two options: `pandas` and `spark`, corresponding to Pandas and PySpark, respectively. Listing 3.3 demonstrates specifying the target code for Pandas.

```
1 ## target code = pandas
```

Listing 3.3: Specifying *target code* as `pandas` for translating subsequent DSL statements into Pandas code.

Data inspection. Subsequently, three options for specifying a dataframe include: "`create dataframe from`" to create a dataframe from a variable, "`load from`" to import a CSV file into a dataframe, and "`on`" to reference a previously defined variable. Listing 3.4 presents a set of DSL commands and corresponding code lines for data inspection with Pandas, after loading a CSV file into a dataframe named `data` (the first three lines). Regular comment lines (beginning with `#` instead of `##`) are provided solely for explanatory purposes.

```
1  # Loading the csv file into dataframe 'data'
2  ## data = load from 'covid_19_data.csv' as csv
3  data = pandas.read_csv('covid_19_data.csv')
4
5  # Describing the dataframe content
6  ## on data | describe | show
7  print(data.describe())
8
9  # Selecting columns 'Country/Region' and 'Confirmed' of the dataframe 'data', and ↩
       ↪ printing the first 10 rows
10 ## on data | select columns 'Country/Region', 'Confirmed' | head 10 | show
11 print(data[['Country/Region', 'Confirmed']].head(10))
12
13 # Grouping rows of the dataframe 'data' by the column 'Country/Region', summing and ↩
       ↪ then sorting by 'Confirmed'
14 ## on data | group by 'Country/Region' apply sum | sort by 'Confirmed' descending | ↩
       ↪ show
15 print(data.groupby(['Country/Region']).sum().sort_values(['Confirmed'], axis= ↩
       ↪ 'index', ascending=[False]))
```

Listing 3.4: Dataframe inspection with Pandas DSL.

Particularly, to inspect a dataframe using Pandas DSL, users can describe the dataframe (lines 5–7) or select and print specific columns/rows (lines 9–11). Additionally, users can perform complex operations such as grouping and sorting (lines 13–15).

In terms of exploring dataframes with PySpark DSL, the DSL statements are identical but the generated PySpark code lines differ slightly from those for Pandas. For simplicity, we omit PySpark code in this section. Additional test cases for all supported DSL (version v0.5.0) are available on our group's website (AIP Group, 2022b).

Data preprocessing. *NLDSL* also delivers operations for commonly used data manipulation tasks. Listing 3.5 exhibits the utilization of Pandas DSL to append a new column named `Active` to the dataframe `data`.

```
# Appending to dataframe 'data' a new column 'Active' which is derived from ' Confirmed', 'Deaths' and 'Recovered'
## on data | append column data.Confirmed - data.Deaths - data.Recovered as 'Active'
data.assign(**{'Active': data.apply(lambda row: row.Confirmed - row.Deaths - row. Recovered, axis=1).values})
```

Listing 3.5: Data preprocessing with Pandas DSL.

DSL Customization

For the last group of core features, vanilla *NLDSL* offers two mechanisms to extend defined DSLs: through expression rules for developers and via internal DSL functions for end-users.

Extending DSLs with expression rules. Developers can enhance and manage DSLs using expression rules, involving three steps: (i) defining the grammar and associated details of the DSL operation, (ii) establishing the expression rule for translating DSL statements into executable code, and (iii) registering the expression rule with the *NLDSL* tool.

Step 1. Grammar documentation. *NLDSL* allows specification of examples, grammar, arguments, and type for a DSL operation/function via a predefined structured string, instead of a conventional doc string. Listing 3.6 represents the string documentation for the operation `group by`, which is displayed alongside the function name during operation-level code completion.

```
1  GROUP_BY_DOC = """Group a DataFrame and apply an aggregation.
2  Examples:
3    1. ## x = on df | group by df.col1 apply min
4    2. ## x = on df | group by df.col1, df.col2 apply mean
5  Grammar:
6    group by $columns[$col] apply $aggregation
7    aggregation := { min, max, sum, avg, mean, count }
8  Args:
9    columns (varlist): A list of column names.
10   aggregation (variable): The aggregation operation to be performed.
11 Type:
12   Operation
13 """
```

Listing 3.6: Documentation defined for DSL operation `group by`, adapted from the source code of vanilla *NLDSL* (Andrzejak et al., 2019a).

Here, `$columns[$col]` and `$aggregation` in the `Grammar` section are two arguments, further specified in the `Args` part. Namely, `$columns[$col]` is a list of variables (or list of columns in this case), while `$aggregation` is a variable with a value defined in a finite set (Line 7). Notably, the `Type` field at the bottom of the string indicates the operation type, such as "`Initialization`" for `Init_Op` or "`Operation`" for `Intermed_Op` and `Final_Op` (depicted in Figure 3.1).

Step 2. Expression rule definition. Given the specified documentation, developers can establish the expression/translation rule for the DSL operation using a concise Python function,

decorated with the term "`@grammar`". This decorator serves as a parser for the DSL operation, utilizing the grammar outlined in the documentation string. The first four lines of Listing 3.7 display the expression rule definition for the `group by` operation.

```
1 @grammar(docs.GROUP_BY_DOC)
2 def group_by(code, args):
3   cols = list_to_string(args["columns"])
4   return code + ".groupby({}).{}()".format(cols, args["aggregation"])
5
6 # Registering the expression rule above
7 PandasCodeGenerator.register_function(group_by)
```

Listing 3.7: Expression rule for DSL operation `group by`, adapted from the source code of vanilla *NLDSL* (Andrzejak et al., 2019a).

The `code` argument denotes the generated code from preceding pipeline operations. Meanwhile, `args` is a dictionary containing arguments parsed from the DSL operation based on the defined grammar. Particularly, `args["columns"]` provides a list of columns specified in the DSL operation, and `args["aggregation"]` returns the token following the keyword "`apply`", indicating the selected aggregation function.

It is worth noting that by this definition, the `group by` DSL operation is always followed by a non-argument `apply` operation, which may not always align with developer preferences. We addressed this issue in the current version of the *NLDSL* extension (details in Section 3.3.1).

Step 3. Rule registration. In the final step, developers must register the defined expression rule with the *NLDSL* tool, as depicted in the last line of Listing 3.7. Here, `PandasCodeGenerator` is a class tailored for Pandas DSL, inherited from the `CodeGenerator` class within the *NLDSL.lib* component, which lays the foundation for code generation functionalities.

Extending DSLs via internal functions. Ultimately, end-users (and also developers) can extend DSLs by defining a new DSL function as a pipeline of existing DSL operations, using the *definition* statement described in Section 3.2.2. The newly defined function can then be utilized as a DSL operation. Listing 3.8 demonstrates an example of using *definition* statement to create a new DSL function named `div columns` (Line 2).

```
1 # Defining a new function named div columns
2 #$ div columns $col1 $col2 as $res = append column $col1 / $col2 as $res
3
4 # Using 'div columns' to calculate 'deathsRatio' from 'Deaths' and 'Confirmed'
5 ## on data | div columns data.Deaths data.Confirmed as 'deathsRatio'
6 data.assign(**{'deathsRatio': data.apply(lambda row: row.Deaths / row.Confirmed,  ↩
    ↪ axis=1).values})
```

Listing 3.8: Defining a new DSL operation via internal function.

The `div columns` function divides two dataframe columns and assigns the result to a specified column (`$res`), which is then appended to the dataframe. This implementation utilizes the

existing `append column` operation. Line 5 shows the usage of the newly defined function, while Line 6 presents the corresponding Pandas code. Notably, upon definition, the new function `div columns` is also included in the recommendation list of the DSL-based Language Server, as depicted in Figure 3.5.

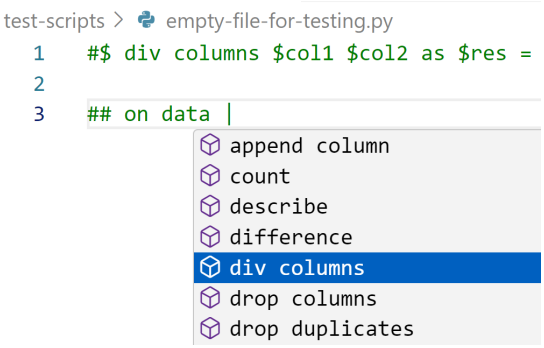


Figure 3.5: Code completion at operation level after defining a new DSL function.

However, this approach encounters three primary challenges: (i) the necessity for users to grasp and become accustomed to existing DSL grammars, (ii) challenges for end-users in implementing a new DSL that may not leverage existing DSL operations, and (iii) the inability to utilize newly defined DSL functions across multiple Python files without re-declaration. To address these concerns, we developed a wizard to streamline the creation of new DSLs using an Excel template, fostering DSL sharing across files and among users. The next section delves into the detailed enhancements made to the *NLDSL* tool.

3.3 NLDSL Visual Studio Code Extension

Constructed upon the foundation of the vanilla *NLDSL*, the deployed *NLDSL* VSCode extension inherits core functionalities (discussed in Section 3.2.3), supplemented with advanced features. This section outlines these enhancements in three categories: DSL development (Section 3.3.1), code completion-related features (Section 3.3.2), and utilities (Section 3.3.3). Advanced features in each category are delineated in Figure 3.6.

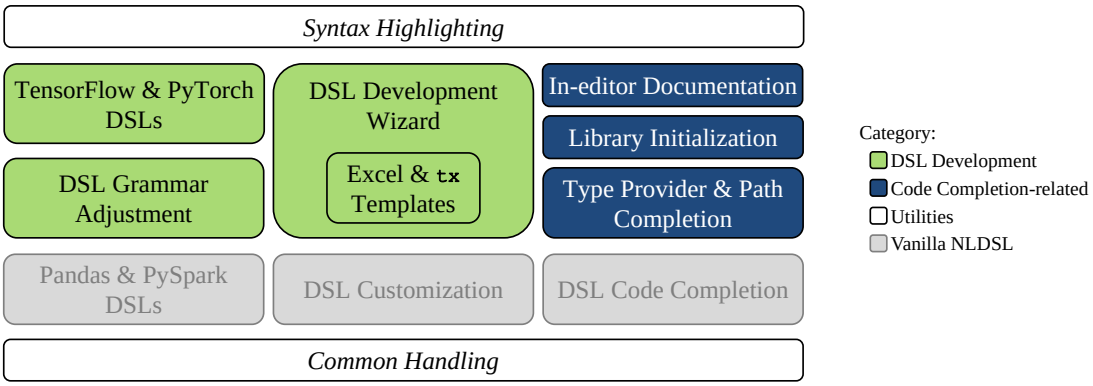


Figure 3.6: An overview of advanced features delivered with *NLDSL* extension.

3.3.1 DSL Development

The *NLDSL* extension augments its predecessor, i.e. the (vanilla) *NLDSL* tool, in facilitating DSL development through various means. Firstly, the extension introduces a wizard for users to create and share DSLs across Python files and users, surpassing the constraints of the base tool. Secondly, two new DSLs (in alpha state) for commonly used DL tasks are incorporated. Ultimately, adjustments to DSL grammars are made to accommodate more complex test cases.

DSL Development Wizard

The DSL wizard is developed within the Bachelor thesis of Waibel (2021) in our research group. Essentially, the wizard enables end-users to manage DSLs in a manner resembling how developers use expression rules (addressed in Section 3.2.3), but without directly accessing the extension's source code. Specifically, the wizard facilitates creation of new DSLs from templates, integration of (defined and shared) DSLs into the extension, and removal of DSLs when needed.

Excel and `tx` templates. Initially, Pfleger (2020) introduced the utilization of `tx` files as templates for DSL generation within our *NLDSL* extension. These `tx` files are then processed by textX to generate the corresponding DSLs. Listing 3.9 exhibits the definition of the DSL operation `group by` in `tx` format.

```
1 FuncModel-Operation-2
2   func: "group_by"
3   grammar: "group by $columns[$col] apply $aggregation"
4   args: "aggregation:min,max,sum,avg,mean,count"
5   syntax: ".groupBy($).$()"
6   type: "Operation"
```

Listing 3.9: Definition for the DSL operation `group by` in `tx` format.

Here, `FuncModel-Operation-2` serves as a unique identifier to differentiate it from other functions/operations registered within the extension. The fields `func`, `grammar`, `args`, `syntax`, and `type` represent the function name, DSL grammar, argument constraints, syntax for translating DSL to executable code, and operation type, respectively. These attributes align with those discussed in Section 3.2.3.

To assist users in adapting to the structure of `tx` files and adhering to the defined constraints for each field, Waibel (2021) introduced an Excel template mirroring the `tx` file structure. This template includes predefined fields with corresponding selectable values in drop-down lists. Figure 3.7 depicts a portion of this Excel template used for defining PySpark DSL. For simplicity, we solely magnify relevant values in the figure.

Namely, values of columns `func`, `grammar`, `args`, `syntax`, and `type` within the Excel file correspond to those defined in the `tx` file for the `group by` operation (Listing 3.9). The values in the `type` column, representing operation types, are pre-specified and presented as a drop-down list upon selection (not shown in Figure 3.7). Detailed instructions and examples for DSL creation are provided on our group's website (AIP Group, 2022a).

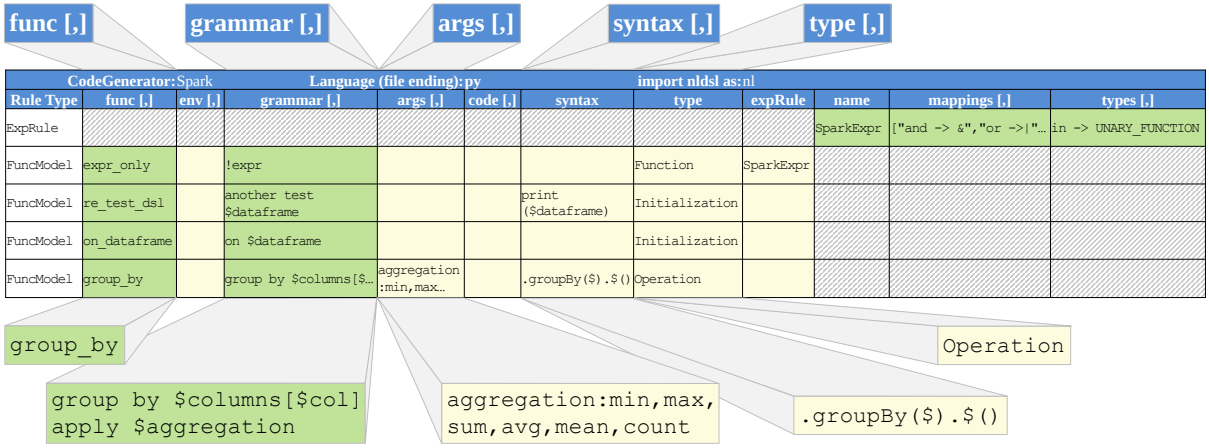


Figure 3.7: An example of Excel template for DSL creation with *NLDSL*.

DSL integration and removal. Users can incorporate a customized DSL into the extension via the *NLDSL* menu, accessed through the side panel of VSCode by clicking on the NL DSL icon. Figure 3.8 displays the *NLDSL* menu after adding a DSL named `example_dsl`. In the list of DSLs, `global` refers to built-in DSLs, while `extern` denotes user-created or shared DSLs.

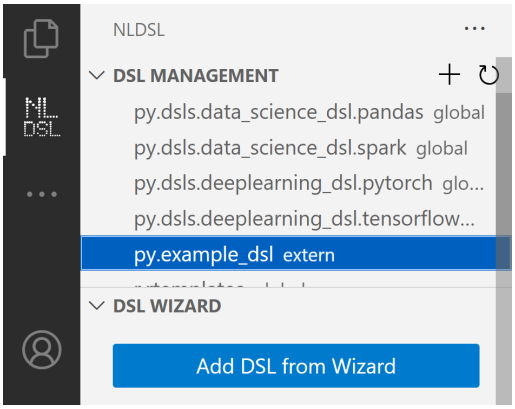


Figure 3.8: *NLDSL* menu in VSCode after adding a new DSL named `example_dsl`.

If users select an Excel file as the input template, the wizard will convert the Excel file into a `tx` file, which is subsequently processed by textX. Upon successfully integrating the new DSL, generated files, such as configuration files and Python files, which contain documentation strings and expression rules, are organized within a DSL folder. Alternatively, users can add a DSL by specifying the path to the DSL folder. Similarly, removing a DSL can also be performed through the *NLDSL* menu. This action will not delete the corresponding DSL folder.

It is important to note that users are not required to perceive the internal procedure of creating and managing DSLs. They simply need to utilize the Excel or `tx` template and remember the paths to their DSL folders. We believe this greatly simplifies DSL development for end-users.

Deep Learning DSLs

To enrich the built-in DSLs shipped with the extension, we supplemented two additional DSLs tailored for PyTorch and TensorFlow frameworks, facilitating standard DL operations. These

DSLs are implemented by our practicum students, Philipp Walz and Jona Neef (Walz et al., 2020). An illustration of TensorFlow DSL usage for basic DL tasks is depicted in Listing 3.10.

```
1  # Creating a sequential model with one flatten layer and one dense layer
2  ## create model myModel | model type feed_forward | add layer flatten | add layer ↩
       ↪ fully connected 36864 128 none
3  myModel = tf.keras.models.Sequential([
4    tf.keras.layers.Flatten(),
5    tf.keras.layers.Dense(128)
6  ])
7
8  # Compiling the model with optimizer adam
9  ## on myModel | compile | optimizer adam
10 myModel.compile(optimizer='adam')
11
12 # Training the model with the train set mnist_train in 10 epoches
13 ## on myModel | train | using mnist_train data with labels | epochs 10
14 myModel.fit(mnist_train[0], mnist_train[1], epochs=10)
15
16 # Evaluate the model with the test set mnist_test
17 ## on myModel | evaluate | using mnist_test data with labels
18 myModel.evaluate(mnist_test[0], mnist_test[1])
```

Listing 3.10: Basic Deep Learning tasks with TensorFlow DSL*.

* DSL statements within the *NLDSL* extension incorporate syntax highlighting (explained later), contrasting with the uniform comment color observed in Section 3.2.

Users define a DL model within the first six lines by specifying the model type (e.g. `feed_forward`) and adding specific layers (e.g. `flatten` and `fully connected`). Here, 36864, 128, and `none` denote the input shape, output shape, and the `softmax` function, respectively. Following model creation, users can compile it (lines 8–10), train it on a designated train set (lines 12–14), and evaluate its performance on a target test set (lines 16–18).

The PyTorch DSL grammar closely resembles that of TensorFlow DSL, facilitating smooth transition between platforms for end-users. However, these operations are still in the alpha stage. Additional examples of TensorFlow and PyTorch usage are accessible via test cases provided on our group's website (AIP Group, 2022b).

DSL Grammar Adjustment

We modified predefined DSL grammars, such as `group by` and `apply`, to accommodate complex test cases. In vanilla *NLDSL*, as detailed in Section 3.2.3, the `apply` operation invariably follows the `group by` operation. Moreover, identifying arguments for the `apply` operation is not feasible, which may not always align with developers' preferences. To address this, we separated the grammar of these operations, redefining the `apply` operation with optional arguments.

Listing 3.11 outlines the updated documentation strings for these operations. The corresponding expression rules are omitted for simplicity. With the revised grammar, Listing 3.12 illustrates the application of `group by` and `apply` operations in the data inspection task, which is previously outlined in lines 13–15 of Listing 3.4. Users can now determine columns for each `apply` operation, along with the resulting column name.

```
GROUP_BY_DOC = """Group a DataFrame and apply an aggregation.
Examples:
  1. ## x = on df | group by df.col1
  2. ## x = on df | group by df.col1, df.col2
Grammar:
  group by $columns[$col]
Args:
  columns (varlist): A list of column names.
Type:
  Operation
"""

APPLY_DOC = """ Apply aggregation function(s) on column(s).
Examples:
  1. ## on data | apply sum on df.col1 as "newColumn"
  2. ## on data | apply sum on df.col1 as "newColumn1", mean on df.col2 as "
    ↪ newColumn2"
Grammar:
  apply $aggregations[$aggregation on $col as $new_col]
  aggregation := { min, max, sum, mean, count }
Args:
  aggregations (varlist): A list of performed aggregations on columns.
Type:
  Operation
"""
```

Listing 3.11: Documentation defined for DSL operations `group by` and `apply` in *NLDSL* extension v0.5.0.

```
# Grouping rows of the dataframe 'data' by the column 'Country/Region', summing
    'Deaths' and then sorting by the column 'Last Update' descending

## on data | group by 'Country/Region', 'Last Update' | apply sum on 'Deaths'  as
    ↪ 'DeathsSummed' | sort by 'Last Update' descending | show

print(data.groupby(['Country/Region', 'Last Update']).agg({'Deaths' : 'sum'}).rename
    ↪ (columns={'Deaths' : 'DeathsSummed'}).sort_values(['Last Update'], axis=
    ↪ 'index', ascending=[False]))
```

Listing 3.12: Usages of `group by` and `apply` operations with adjusted grammars.

3.3.2 Code Completion-related Features

Besides the fundamental code completion at operation and statement levels (explained in Section 3.2.3), the *NLDSL* VSCode extension offers additional features, augmenting the code completion mechanism. These supplements involve type provider and path completion for CSV files, library initialization through *target code* declaration, and in-editor documentation.

Type Provider and Path Completion

As CSV files are commonly used in data analysis, recognizing file structure and providing relevant information would benefit end-users. Weber (2020), in his Bachelor thesis at our research group,

developed type provider to recommend column names when handling CSV files, implemented path completion to suggest paths of available CSV files within the project folder, and introduced syntax highlighting to enhance readability (discussed later in Section 3.3.3).

Type provider. The idea behind this feature involves extracting structural information from a CSV file, including column names, row values, and column types, to construct metadata. The metadata is then used to form a key-value dictionary representing supported options of the type provider. For instance, keys like `columns` or `col` contain the column names extracted from the CSV file. These keys also align with arguments defined in DSL grammar (e.g. `$columns` for the `select columns` operation).

Subsequently, based on the ongoing operation, relevant values are injected into the extension's completion results before presenting the recommendation list to users via the language server. To enable this functionality, users must initially specify the desired CSV file using a DSL statement prefixed with `#!`, as demonstrated in Listing 3.13.

```
1 # Specifying CSV file 'data/covid_19_data.csv' for the type provider
2 #! name 'data', type 'csv', path 'data/covid_19_data.csv'
```

Listing 3.13: Specifying a CSV file for type provider feature.

Here, `name` denotes the dataframe's identifier for subsequent data processing. The terms `type` and `path` specify the file's format (e.g. `'csv'`) and its location, respectively. After setting the metadata for the type provider, *NLDSL* can suggest column names from the specified CSV file. Figure 3.9 illustrates the recommended column names for the `select columns` operation upon activating the type provider.

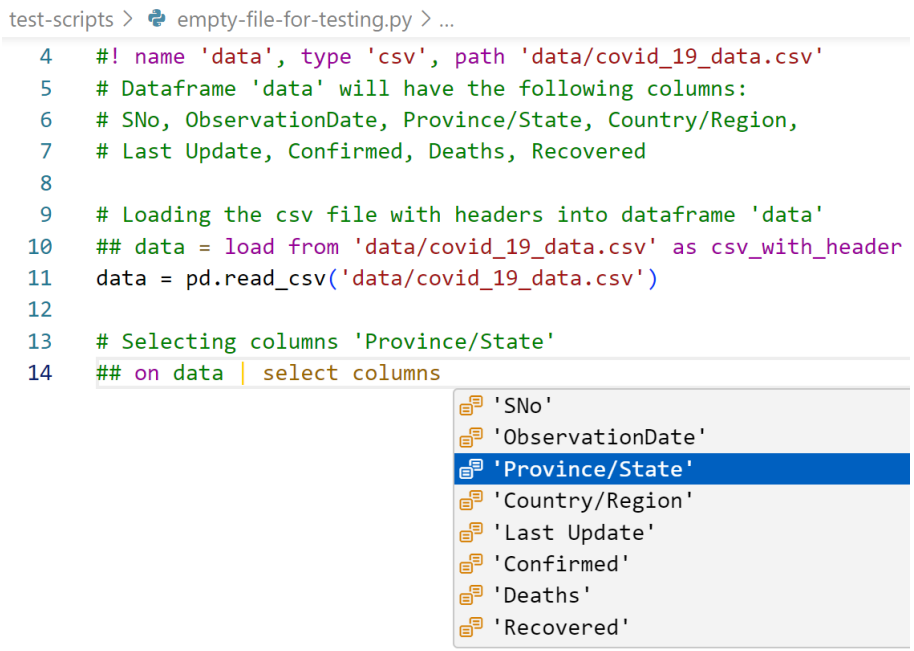


Figure 3.9: Column names completion with type provider in *NLDSL**.

* The `load from` operation differs slightly from that in Listing 3.4 due to adjustments in the DSL grammar.

Path completion. Using the same concept, paths to all files within the project directory are gathered and stored in the aforementioned key-value dictionary with the key named `path`. This list of file paths is then integrated into the completion results when the `$path` argument is expected next in the DSL operation. Figure 3.10 demonstrates path completion for the `load from` operation.

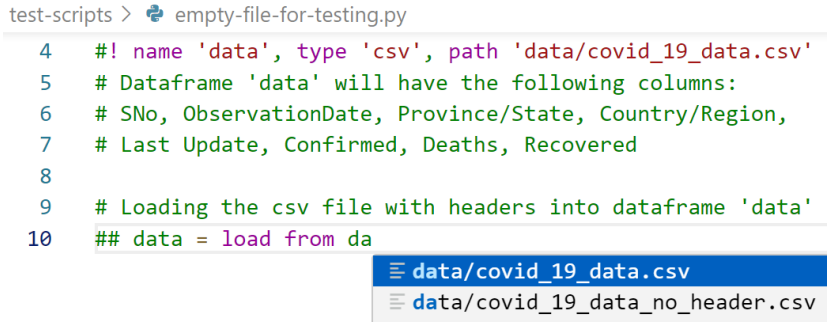
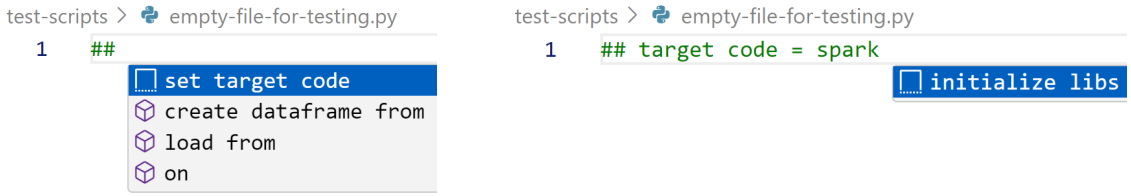


Figure 3.10: Path completion supported by *NLDSL*.

Library Initialization

The next supplemented feature involves automatically importing specific Python libraries to ensure the executability of code translated from DSL statements. For certain cases like Pandas or TensorFlow DSLs, the import statements can be inferred (i.e. "`import pandas`" or "`import tensorflow`"). However, each *target code* often requires particular libraries and sometimes instantiation (e.g. PySpark DSL), which might be unfamiliar to end-users.

To tackle this issue, we introduced a DSL function, "`set target code`" (Figure 3.11(a)), facilitating users in specifying their *target code*. Moreover, upon selecting the completion option "`initialize libs`" after defining the *target code* (Figure 3.11(b)), relevant libraries will be automatically imported. Listing 3.14 displays examples of generated codes importing relevant libraries for PySpark and PyTorch.



(a) (b)

Figure 3.11: Setting *target code* and initializing libraries with *NLDSL*.

```
## target code = spark
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
spark = SparkSession.builder.appName("Spark Example").getOrCreate()

## target code = pytorch
import torch as pyTorch
import torchvision
```

Listing 3.14: Initializing relevant libraries for `spark` *target code* in *NLDSL* extension.

Specifically, users can customize the libraries for each target code using a JavaScript Object Notation (JSON) configuration file shipped with the extension. Listing 3.15 illustrates a portion of this configuration file. Furthermore, in the current version of the *NLDSL* extension (v0.5.0), we enhanced the underlying mechanism to enable users to specify multiple target codes within a Python file, as depicted in Listing 3.14. This eliminates unnecessary file switches due to changes in *target codes*, a feature lacking in the original *NLDSL* tool.

```
"auto_import": {
  "pandas": "\nimport pandas as pd\n",
  "spark": "\nfrom pyspark.sql import SparkSession\nfrom pyspark.sql.functions
    ↪ import *\n\nspark = SparkSession.builder.appName(\"Spark Example\").
    ↪ getOrCreate()\n",
  "pytorch": "\nimport torch as pyTorch\nimport torchvision\n",
  "tensorflow": "\nimport tensorflow as tf\n"
}
```

Listing 3.15: Configuration of auto-imported libraries for each *target code*.

In-editor Documentation

Finally, to complement the code completion aspect, we implemented an in-editor documentation feature. Despite the DSL development wizard (Section 3.3.1) assisting users in creating DSLs without directly engaging with the extension's source code, users still need to remember the grammar of defined DSLs. Hence, we deployed a completion option called "`Help on current command`", accessible at the bottom of the completion list, to aid in recalling DSL grammar for the current command during editing. Figure 3.12 presents the case for the `apply` operation.

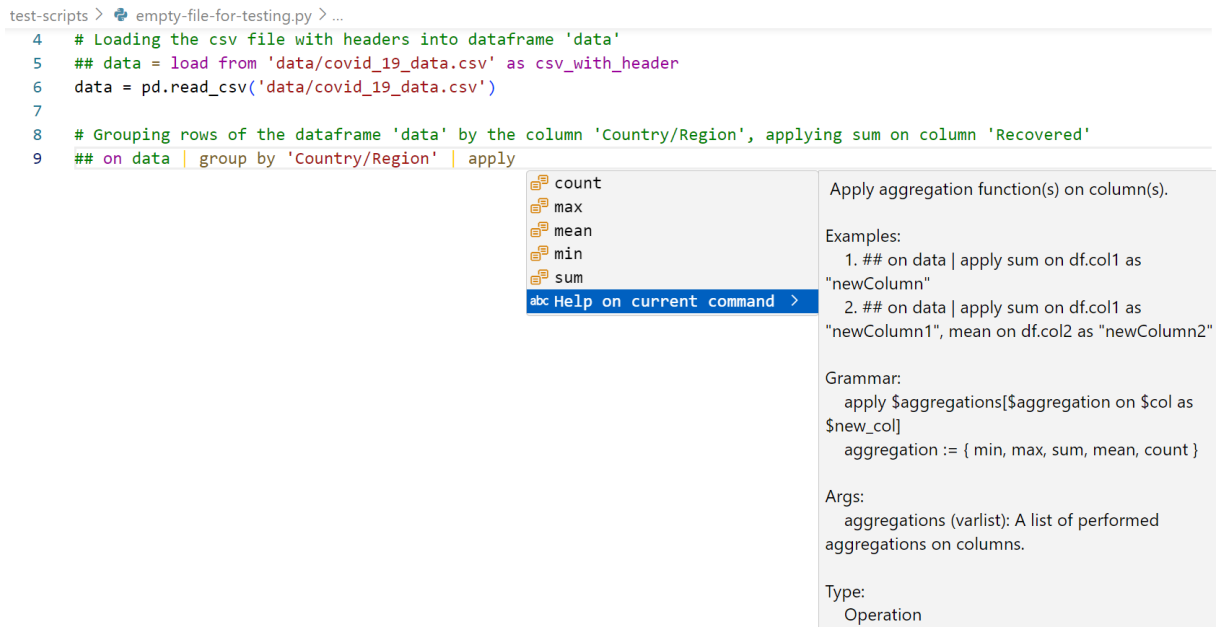


Figure 3.12: Recalling DSL grammar for `apply` operation while editing with *NLDSL*.

The current command's documentation is obtained by comparing keywords of the incomplete operation (from the pipe symbol to just before the cursor) with defined DSL grammars. For simplicity, the documentation string of the first operation in the search results is retrieved.

3.3.3 Utilities

The last category of advanced features introduced to the *NLDSL* extension, compared to the vanilla tool, includes common handling and syntax highlighting. The former facilitates extension maintenance for developers, while the latter improves readability and user experience.

Common Handling

We redesigned the extension's workflow to centralize control over additional features such as type provider, path completion, and synchronization indicator (discussed later). These features are managed through handlers, registered via a JSON configuration file. This approach enables developers to integrate new features into the extension by adding the corresponding handler to the configuration file, without altering the extension's source code.

Apart from code completion, the common handling procedure currently supports four event types: `did_open`, `did_change`, `did_save`, and `did_close`, signifying user actions of opening, modifying, saving, and closing a Python file, respectively. Furthermore, lines within the Python file are categorized into three groups: `code` for code lines, `dsl` for DSL statements, and `directive` for DSL directives. In the current version (v0.5.0), DSL directives specify CSV files for the type provider feature, identified by lines starting with `#!`.

For each line type within an event, corresponding handlers are designated in the configuration file. Listing 3.16 outlines the handlers registered for the `did_change` event (lines 9–23), namely `type_provider_handler` and `syncer_handler`.

```
1  "handlers": {
2    "syncer_handler": {...},
3    "type_provider_handler": {
4      "module": "nldsl_plugin.TypeProvider.type_provider_handler",
5      "class": "TypeProviderHandler",
6      "sub_config": {...}
7    }
8  },
9  "did_change": {
10   "code_dsl": {
11     "syncer_handler": {
12       "instance": "syncer_handler",
13       "functions": ["did_change"]
14     }
15   },
16   "directive": {
17     "syncer_handler": {...},
18     "type_provider_handler": {
19       "instance": "type_provider_handler",
20       "functions": ["did_change"]
21     }
22   }
23 }
```

Listing 3.16: Handlers registered for `did_change` event in *NLDSL*.

Here, `code_dsl` denotes that the handler is utilized for both `code` and `dsl` line types. Besides,

`instance` refers to an instance of the handler class defined previously (lines 1–8), while the field `functions` determines the names of functions to be performed with the handler. When the *NLDSL* extension is activated, all handler classes identified in the configuration file are instantiated and registered for the four aforementioned events. Depending on the event and encountered line type, the handlers execute associated functions to process the code/DSL line.

Syntax Highlighting

Weber (2020), in his Bachelor thesis, improved DSL readability through syntax coloring and introduced a synchronization indicator to detect mismatches between DSL lines and code lines. We integrated this work into the extension, which has been available since version 0.1.6.

Syntax coloring. The syntax coloring feature operates on the client side of the language server, implemented in TypeScript. End-users can customize color rules seamlessly via a JSON configuration file provided with the extension. In the current version (v0.5.0), colors are determined by the VSCode theme (e.g. light or dark), token types (e.g. keyword, string, or number), and DSL operation type (e.g. initialization or normal operation). Examples of colored DSL statements are illustrated through listings and figures in Section 3.3.

Synchronization indicator. An inherent issue with *NLDSL* (v0.5.0) is that updating the DSL statement may not automatically update the corresponding code line if it has already been generated, potentially resulting in incorrect execution results. To mitigate this, we integrated a synchronization indicator into the extension to alert users when these two lines are out of sync.

This indicator leverages the diagnostic publishing feature of the language server implemented with pygls. Upon encountering an event (e.g. `did_open` or `did_change`), a warning diagnostic (`severity = 2`) is created if there is a mismatch between a DSL statement and its related code line. These two lines are in sync if the code line aligns with the code translated from the DSL statement to the specified target language. Listing 3.17 exhibits an example of unsync cases.

```
1 # Loading a csv file without headers into dataframe 'data'
2 ## data = load from 'data/covid_19_data_no_header.csv' as csv_without_header
3 data = pd.read_csv('data/covid_19_data.csv')
```

Listing 3.17: Synchronization indicator with *NLDSL*.

We anticipate that the robust foundation of the vanilla *NLDSL* tool, combined with enhancements in the VSCode extension, will augment DSL development and usage for both end-users and developers. The next section outlines our dissemination strategy to make the extension more accessible to users.

3.4 Dissemination

The *NLDSL* VSCode extension is deployed and managed through Microsoft Azure Pipelines. We utilized Continuous Integration (CI) and Continuous Deployment (CD) pipelines to build and publish multiple versions of the extensions, respectively. This is attributed to the parallel job support, across diverse Operating System (OS) images, on Azure.

The extension is delivered on VSCode Marketplace in three versions for Windows, Linux and macOS[4]. These versions share the extension's source code but differ in the configuration files, i.e. YAML files, which define necessary steps in building the extension. Notably, a standalone Python interpreter is included with the extension, allowing users to utilize *NLDSL* without installing Python on their devices.

Figure 3.13 depicts an overview of our steps for implementing the dissemination, entailing repository and Azure account setup, YAML file creation, CI pipeline configuration, and CD pipeline design. This section describes our approach to manage CI and CD pipelines, followed by detailed build instructions specified in YAML files. Comprehensive guidance for publishing a VSCode extension can be found in Azure's documentation[5].

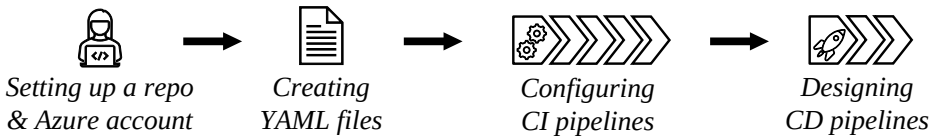


Figure 3.13: General steps for disseminating the *NLDSL* extension with Azure Pipelines.

3.4.1 Managing CI and CD Pipelines on Azure

Microsoft (2023) proposes two methods for utilizing CI and CD pipelines. The first approach involves setting up a single YAML file for both pipeline types, automatically triggering the CD pipelines immediately after the CI ones. Alternatively, developers can use the classic Azure DevOps web portal for manual activation. We have adopted the latter for our *NLDSL* extension.

Particularly, we established CI pipelines using predefined YAML files, acquired built artifacts, and subsequently defined CD pipelines to consume and deploy these artifacts. This enables testing of artifacts before the deployment, particularly for features involving user interaction, such as the DSL development wizard. Figure 3.14 displays the workflow triggered during the release of new extension features.

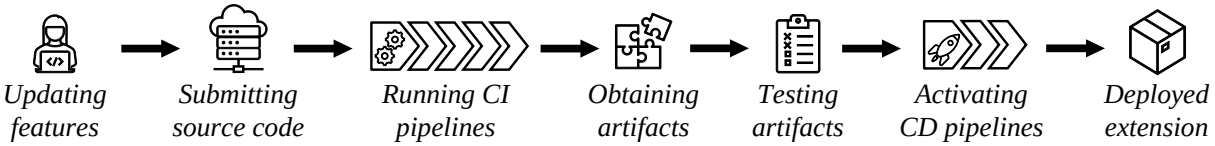


Figure 3.14: Workflow of activating CI and CD pipelines while releasing new extension features.

After updating new features, developers submit source code to the repository, prompting CI pipelines to build their artifacts. These artifacts can be downloaded from the Azure DevOps web portal and manually tested with VSCode. Subsequently, the CD pipelines are activated to publish the tested artifacts, making the extension available on the Marketplace for users.

[4] Visual Studio Marketplace for *NLDSL* extension on three different operating systems, `https://marketplace.visualstudio.com/publishers/PVS-IfI-Heidelberg-University-Germany`.

[5] Azure Pipelines documentation, `https://learn.microsoft.com/en-us/azure/devops/pipelines`. (Access: 05 March 2024).

Given that managing CI and CD pipelines is conducted through Azure web interface with predefined options, the primary task is to specify necessary steps in the YAML file for generating CI artifacts. The next subsection delves into build instructions designated in the YAML file.

3.4.2 Build Instructions in YAML Files

In CI/CD, Yet Another Markup Language (YAML) files serve as configuration files defining jobs, stages, and steps for application or service building, testing, and deployment. These files enable developers to specify dependencies, environment variables, and other parameters essential for pipeline execution[6].

We employed three YAML files to configure CI pipelines for three variants of the extension, separately. While these files differ in OS images and scripting languages used for crawling the Python interpreter (e.g. PowerShell for Windows, Bash for Linux and macOS), they share the common tasks as demonstrated in Figure 3.15.

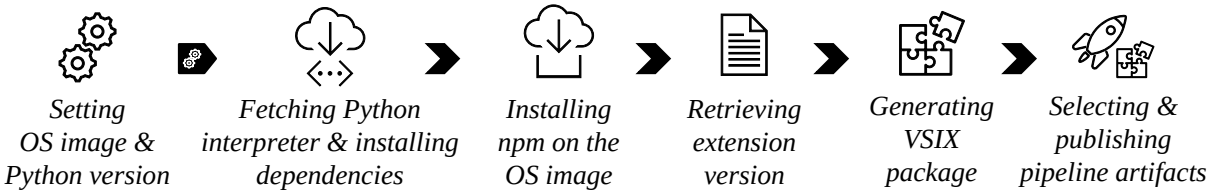


Figure 3.15: Tasks specified in YAML files for CI pipelines.

Following the configuration of the target OS image and Python version (e.g. `windows-latest` and Python 3.7 for Windows), a standalone Python interpreter is fetched into a directory bundled with the extension, along with the installation of relevant dependencies. Subsequently, Node Package Manager (npm) is installed on the OS image to mange the packing process.

The version of the extension is then retrieved and used during the generation of the VSIX package. This package is a `.vsix` file containing one or multiple Visual Studio extensions and the accompanying metadata utilized by Visual Studio for classification and installation of the extensions (Microsoft, 2022). Ultimately, the resulting VSIX package and extension version (stored in a text file) are moved to a staging directory (`Build.ArtifactStagingDirectory`) and published as a pipeline artifact for later use in the CD pipelines.

It is worth mentioning again that we performed the testing phase, on both the extension's source code and the obtained `.vsix` files, independent to the setting in the YAML files. This arises from the necessity of user interaction in some features of our extension, such as the DSL development wizard.

Particularly, Listing 3.18 outlines general steps to construct a Python interpreter for the Linux version of the extension. Typically, we downloaded a Python build standalone file, extracted it to the target folder, rename the folder based on the select OS image, and installed relevant extension repositories. Similar scripts for Windows and macOS follow analogous principles but vary in commands used. For simplicity, these scripts are excluded from our discussion.

[6]What is YAML?, `https://www.jetbrains.com/teamcity/ci-cd-guide/faq/yaml/`, (Access: 05 March 2024).

```
echo "Download Python build standalone"
wget "link-to-build-standalone-file" -O python-linux.tar.zst
echo "Extract python tar.zst file"
sudo apt install zstd
zstd -d python-linux.tar.zst
tar -xvf python-linux.tar
rm python-linux.tar.zst
rm python-linux.tar
echo "Rename folder python to Linux"
mv python Linux
echo "Install relevant repos"
cd Linux/install/bin
./python3 -m pip install git+https://relevant-repo-links
```

Listing 3.18: Setting up a standalone Python interpreter for the Linux-*NLDSL* extension.

Our extension's source code, the aforementioned YAML files, and three scripts for fetching Python interpreters are openly accessible through the repository specified on the extension's Marketplace web page (mentioned above). The subsequent section presents our preliminary evaluation for the extension and potential enhancements for future versions.

3.5 Discussion

A thorough evaluation of our extension requires a carefully designed user study with a sufficient participant pool. However, constrained by time and the scope of our research, we defer this analysis to future work and instead concentrate on evaluating the efficacy of utilizing the *NLDSL* extension for fundamental data science tasks. This section presents our preliminary assessment of the extension and offers potential recommendations for future enhancements. The section concludes with our response to the CRQ1, specified in Section 1.4 of Chapter 1.

3.5.1 Preliminary Evaluation

We conducted a preliminary assessment of the *NLDSL* extension across several dimensions, including (i) its assistance for developers in creating DSLs and the coverage in basic data science tasks of supported DSLs, (ii) community reaction measured by installation numbers via VSCode, and (iii) an analysis of the extension's strengths and weaknesses.

Evaluation of the Core Libraries, Supported DSLs, and Advanced Features

> The deployed *NLDSL* extension empowers developers in defining DSLs through concise Python functions, with delivered DSLs capable of covering most essential data exploration and preprocessing tasks. In addition, its advanced features execute in approximately one second on notebooks with moderate computational capability.

Core libraries. In particular, developers can construct and manage DSLs using terse doc strings (e.g. Listing 3.6) and concise Python functions defining expression rules of translating DSL operations to target languages (e.g. Listing 3.7). For instance, the Python files detailing expression rules for Pandas and PySpark DSLs consist of 175 and 200 lines of code (LOC) respectively, excluding blank and comment lines. The documentation file, which can be shared between these two DSLs due to their identical sets of operations, contains 267 non-empty lines.

For end-users, a DSL command pipeline generates one to multiple lines of Python code (e.g. library initialization). Furthermore, these DSL operations are not only designed in a natural-like language, enhancing comprehension for end-users, but also slightly reduce the average number of characters required for typing, compared to the corresponding Python code.

As an illustration, in our Pandas DSL test cases (AIP Group, 2022b), the average length of a DSL pipeline is 59.5 characters, while the average length of corresponding Python code is 63.7 characters (i.e. a 6.6% decrease). For PySpark DSL, these values are 58.4 and 61.3 characters, respectively, resulting in a 4.7% decrease.

Supported DSLs. Besides, Andrzejak et al. (2019a) assessed that the original DSLs for Pandas and PySpark cover 12 out of 14 (85.8%) of Pandas processing steps and 14 out of 16 (87.5%) of PySpark processing steps from a popular DataCamp tutorial for Apache Spark[7]. The current Pandas and PySpark DSLs have been evolved on the basis of the original versions, while maintaining the same coverage. Besides, although TensorFlow and PyTorch DSLs are in the alpha stage, they already support fundamental DL tasks such as dataset loading, model creation, training, and evaluation. Test cases for each predefined DSL are available on our group's website (AIP Group, 2022b).

DSL development wizard. For the advanced features of the *NLDSL* extension, Waibel (2021) demonstrated in his thesis that with 22 DSL operations defined for PySpark, the DSL development wizard can generate necessary files implementing the DSL from an Excel template in less than one second on an i5 4x1.6GHz notebook. The runtime naturally increases as the number of defined operations in the Excel file grows (e.g. around 10 seconds for 100 operations), primarily due to accessing and converting the Excel file to a `tx` file. However, it is important to note that this transformation is performed only once, independent of the DSL usage.

Type provider. Additionally, Weber (2020) found that generating metadata for the type provider feature from a CSV file with 31 columns, 10,000 rows, and an approximate size of 1MB takes around 1.67 seconds on an i7 4x2.7GHz processor. To reduce this processing time, users can set a row threshold in the extension's configuration file, allowing a sample of the CSV file to be handled instead of the entire file.

Syntax highlighting. Ultimately, Weber (2020) also estimated that coloring 1,000 DSL lines takes around 200 milliseconds. For the synchronization indicator, the *NLDSL* extension checks only the line that triggered the `did_change` event and disregards preceding lines, thus saving time compared to examining all existing lines within the file. For instance, in a Python file with 200 DSL lines and one unsynchronized case, the detection time is approximately 14 milliseconds.

[7]Apache Spark Tutorial: ML with PySpark, `https://www.datacamp.com/community/tutorials/apache-spark-tutorial-machine-learning`, (Accessed: 08 March 2024).

It is worth highlighting that the computers used in the above evaluation were modest compared to current high-end configurations[8]. The preliminary evaluation results demonstrate acceptable performance of the *NLDSL* extension on supported features. However, a proper evaluation should include comparisons with similar extensions and consideration of user feedback regarding the extension's utility. The former is currently challenging due to the unique attributes of our extension, but we intend to address the latter in future project work.

Statistics on Dissemination

> After nearly four years of deployment, our *NLDSL* extension has received encouraging community feedback, as indicated by installation numbers.

Over 17k installations. Figure 3.16 depicts a stacked area chart illustrating installation trends for different variants of the *NLDSL* extension, sourced from VSCode Marketplace reports, spanning from July 2020 to July 2024. As of 11 July 2024, the extension has accumulated a total of **17,300** installations, with 9,699 on Windows, 3,829 on macOS, and 3,772 on Linux.

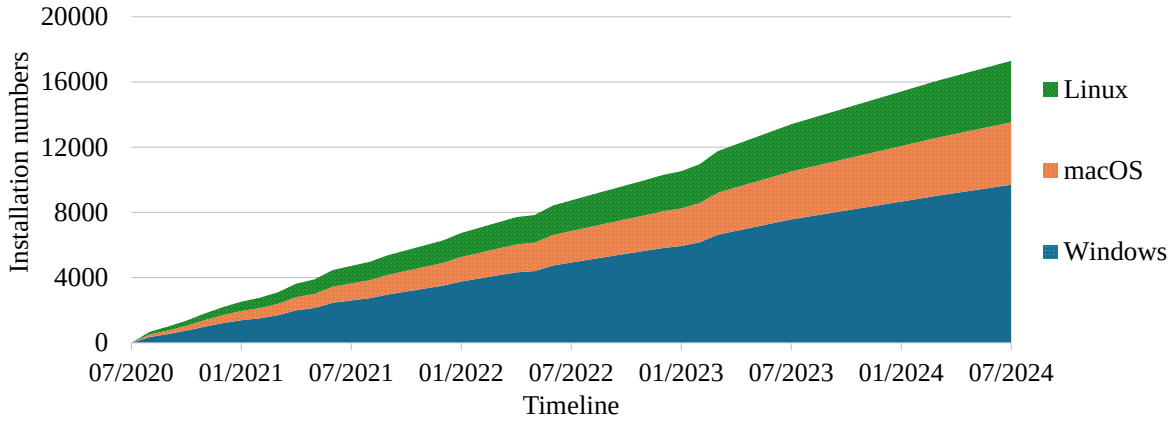


Figure 3.16: Statistics on installation numbers of three variants of *NLDSL* extension over time.

Furthermore, over the last 90 days as of 11 July 2024, the Windows variant recorded 503 installations, while the macOS and Linux variants reached 183 and 212 installations, respectively. Overall, installation numbers have shown a gradual increase over the years, with the Windows variant being the most favored. Additionally, total installations on macOS slightly surpass those on Linux. While these figures may not rival those of extensions published by major corporations, they remain encouraging for our project.

Listed in the top-3 search results. Moreover, our *NLDSL* stands out among extensions available on the VSCode Marketplace for its unique support of DSL usage and development. Meanwhile, other extensions primarily focus on assisting a single specific DSL, such as *C4 DSL*[9],

[8] Fastest CPU of 2024, https://www.techradar.com/pro/fastest-cpus-of-year.
[9] Deployed by Systemticks, https://marketplace.visualstudio.com/items?itemName=systemticks.c4-dsl-extension.
(Accessed: 09 March 2024).

Structurizr DSL[10], and *Asset Management Engine DSL*[11]. Finally, when searching for extensions using terms like "*Data analysis DSL*" and "*DSL development*" on the Marketplace, our extension consistently appears within the top-3 results, regardless of sorting by relevance or trending.

Strength and Weakness of the NLDSL Extension

Although the *NLDSL* extension resolves certain deficiencies of *low-code* approaches (e.g. lack of customization and *vendor lock-in*), it exhibits both 👍 advantages and 👎 disadvantages, indicating areas for refinement.

> 👍 The *NLDSL* extension facilitates usage and development of DSLs for both developers and end-users, tackling the issues of *programming barrier* </> , *reuse* 🔄, and *scalability* 📈 (defined in Section 1.1 of Chapter 1).

DSL usage. As detailed in Sections 3.2 and 3.3, 👍 the extension enables the use of DSLs directly within GPL files as comments, mitigating complexities associated with tool switching. Furthermore, 👍 it offers code completion at both DSL operation and statement levels to aid users in constructing the DSL pipeline and translating commands into executable code.

👍 Advanced features including type provider and path completion enhance code completion by supplying column names, dataframe names from CSV files, and file locations within the project directory. Moreover, 👍 in-editor documentation provides access to DSL grammars during editing, and library initialization supports users to import and initialize relevant libraries by specifying the target language (i.e. *target code*). These completion features, combined with the natural-like language of the DSLs, aim to address the *programming barrier* </> for end-users.

Additionally, among delivered DSLs of the *NLDSL* extension, 👍 the Pandas and PySpark DSLs employ identical operation sets, allowing seamless transitioning between sequential and parallel processing modes. Similarly, 👍 TensorFlow and PyTorch DSLs share common grammars, enabling users to switch between platforms effortlessly. This concept targets to alleviate the challenge of *scalability* 📈.

DSL development. Furthermore, 👍 with *NLDSL*, developers can create and manage DSLs using documentation strings and short Python functions, while end-users can utilize internal functions or a wizard with Excel or `tx` templates, independently of the extension's source code. 👍 In addition, users can share their designed DSLs by distributing the specified template files, lessening the *reuse problem* 🔄.

Ultimately, 👍 feature control in the extension is centralized to configuration files, assisting developers in implementing new features or maintaining existing ones. 👍 Syntax coloring and synchronization indicators are also deployed to enhance the readability of DSL commands. However, there is still room for improvement.

[10] Deployed by GFR Software, https://marketplace.visualstudio.com/items?itemName=gfrsoftware.structurizr-dsl-abacus-extension.

[11] Deployed by Nova Energy Consulting, https://marketplace.visualstudio.com/items?itemName=nova-energy-consulting.amedsl-lang. (Accessed: 09 March 2024).

> 👎 Extension configuration remains constrained to JSON files, while creating complex DSLs with the Excel template might encounter certain challenges.

The *NLDSL* extension provides configuration options through various JSON files. For example, file `handlers.json` is used to specify handlers for events and features, while files `coloring_data.json` and `coloring_colors_light.json` (or `coloring_colors_dark.json` for VSCode dark theme) are employed to adjust syntax coloring. 👎 To update these files effectively, users should understand the file structures and value constraints, potentially adding complexity to the extension usage.

Additionally, the Excel template serves as an initial design for the DSL development wizard. 👎 Complex DSLs with numerous operations and lengthy expression rules could hinder users' ability to quickly grasp the DSL overview as the Excel file expands. Furthermore, 👎 users can only validate the functionality of their defined DSLs after creation and integration into the extension (Waibel, 2021). This might prolong the process of DSL customization.

In the following subsection, we delineate possible upgrades to address the aforementioned limitations and augment the extension with further advanced features.

3.5.2 Potential Enhancements

> ℹ️ Future work for the *NLDSL* extension encompasses various aspects, spanning from editor functionality, DSL utilization, code completion to user experience.

Enriching editor functionality. Firstly, ℹ️ utilizing a more user-friendly display, such as VSCode's setting editor[12], could effectively address the JSON configuration file issue mentioned above. Besides, the current extension allows users to edit DSLs textually, which might be less preferable than modifying them using tables, diagrams, or forms with a projectional editor (Fowler, 2010). ℹ️ A promising development involves a *hybrid editor* integrating textual and projectional editing. For instance, Korz et al. (2023) proposed augmenting a textual code editor with embedded textual or graphical GUIs for code segments. The latter demonstrates its benefit in displaying mathematical formulas.

Enhancing DSL accessibility. Next, ℹ️ the *NLDSL* extension could adopt a web platform or server for users to exchange customized DSLs, enabling upload, sharing, and download within the community. In addition, ℹ️ to mitigate the issue of Excel templates expanding with complex DSLs, other templates combining a `tx` file and a GUI within VSCode could be explored. Notably, these templates should integrate a checker to validate grammar of expression rules before integrating the DSL.

Improving code completion. Subsequently, despite the feature of in-editor documentation in recalling DSL grammar during editing, users still need to select this option from the recommendation list and filter relevant information from the displayed documentation. ℹ️ A potential

[12] User and workspace settings, `https://code.visualstudio.com/docs/getstarted/settings`, (Accessed: 10 March 2024).

enhancement could comprise a *phantom* recommendation, automatically generating complete DSL grammars with hinted parameters to indicate places for parameter names.

For instance, "`select columns <col names>`" could be suggested to prompt users to input column names after the "`select columns`" keyword. This feature should complement type provider and path completion features. Notably, leveraging LLMs to enhance code completion for the *NLDSL* extension requires a sufficient amount of training or fine tuning data, which might currently be unattainable due to the novelty of the proposed DSLs.

Expanding IDE and GPL Support. Alternatively, an enhancement could be extending the extension to IDEs supporting Microsoft's LSP, such as JupyterLab and PyCharm, given that the extension was developed with a DSL-based Language Server. Additionally, adapting our concept to other GPLs like JavaScript and R is also feasible.

Increasing automation. Another potential improvement is to automatically update the executable code when users modify the DSL pipeline, or vice versa. This would reduce the occurrences of unsynchronized cases arising from user editing mistakes.

Incorporating user feedback. Last but not least, user perspectives play a vital role in the iterative process of application development. Therefore, conducting surveys to gather user input is essential for enhancing the extension with practical features.

3.5.3 Response to CRQ1

The first core research question (CRQ1), outlined in Section 1.4 of Chapter 1, focuses on the benefits of embedded *external* DSLs in implementing data analysis tasks. The results from the preliminary evaluation above consolidate our response to this question. Particularly, building upon the acknowledged advantages of DSLs in accelerating programming (analyzed in Section 2.1.2), our concern shifts towards streamlining DSL development for both developers and end-users, a challenge addressed by our proposed *NLDSL* extension.

The preliminary evaluation (Section 3.5.1) emphasizes the advantages of the *NLDSL* extension in enhancing the utilization and development of DSLs for end-users and developers, with features processed in an acceptable duration (approximately one second or less). Moreover, the extension has achieved positive reaction from the community. Certain shortcomings of the extension can be tackled technically with future work, as outlined in Section 3.5.2.

In summary, the *NLDSL* extension exemplifies the effectiveness of embedding *external* DSLs into GPLs as comments to facilitate programming for both domain experts and practitioners. Designed DSLs shipped with the extension cover fundamental data analysis, data processing, and DL tasks. Furthermore, advanced features of the extension also demonstrate the ability to simplify DSL development across various domains. Consequently, we can respond to the first core research question as follows:

> **A-CRQ1. The embedded *external* DSLs exhibit high potential in expediting programming for data analysis tasks and beyond.**

3.6 Summary

In this chapter, we described *NLDSL*, a VSCode extension aimed at facilitating DSL utilization and development for both end-users and developers. The extension is freely available on the VSCode Marketplace, with versions for Windows, macOS, and Linux. Our preliminary evaluation of the extension's core libraries highlights their strength to empower developers and end-users to define DSLs through concise Python functions and dedicated wizards with Excel and `tx` templates, respectively.

Additionally, processing times for enhanced functionalities of the extension are estimated to be roughly one second or less on notebooks with average computing resources, thus expanding the extension's applicability. Specifically, dissemination statistics underscore positive response from the community, spurring further developments, such as editor functionality enrichment, DSL accessibility enhancement, and user feedback integration.

Besides, through the utilization of DSLs along with unique designs and advanced features, the *NLDSL* extension helps mitigate the challenges of *programming barrier* </> , *reuse problem*, and *scalability problem*. These are the primary issues targeted in this dissertation, as outlined in Chapter 1, Section 1.1. Furthermore, the extension illustrates the efficacy of embedded *external* DSLs in accelerating programming, addressing the first core research question (CRQ1), specified in Chapter 1, Section 1.4. This is also a representation of our practical contributions to the dissertation.

Ultimately, the *NLDSL* extension entails work from our practicum and Bachelor students. Table 3.1 details the features and tasks involved in the process of developing and distributing the extension, along with the corresponding contributors. The subsequent three chapters delineate the outcomes of our research contributions.

Table 3.1: Contributors in developing and disseminating the *NLDSL* extension.

Features/Tasks	**Section**(s)	**Contributors**	**Source(s)**
Vanilla *NLDSL*	3.2.3	Oliver Wenz, Oliver Schmitt, Kevin Kiefer, Diego Elias Costa, Artur Andrzejak	Andrzejak et al. (2019a), Wenz (2019), Schmitt (2019)
Type provider, Syntax highlighting	3.3.2 3.3.3	Patrick Weber	Weber (2020)
Deep learning DSLs	3.3.1	Philipp Walz, Jona Neef	Walz et al. (2020)
DSL development wizard	3.3.1	Tim Waibel, Dennis Pfleger	Waibel (2021), Pfleger (2020)
Library initialization, In-editor documentation, Common handling, Feature integration, Dissemination	3.3.2 3.3.2 3.3.3 - 3.4	Kim Tuyen Le	This dissertation
DSL grammar adjustment, Extension testing, Bug fixing, Documentation	3.3.1 - - -	Kim Tuyen Le, Christopher Höllriegl	This dissertation AIP Group (2022b) - -
Extension testing, Project supervision	-	Artur Andrzejak	-

Part III

Research Contributions

Extended Network

Chapter 4

Improving Code Recommendations by Combining Neural and Classical ML Approaches

Our first work on the research side focuses on code recommendation systems, which target to expedite scripting tasks and development of large-scale software projects. Widely adopted features of these systems include code completion and next token prediction provided by modern Integrated Development Environments (IDEs). However, dynamic languages like Python present a notable challenge for these systems due to the scarcity of type information during editing.

Various Machine Learning (ML) approaches have been proposed to address this challenge. Particularly, the Probabilistic Higher Order Grammar (PHOG) technique suggested by Bielik et al. (2016) utilizes a grammar-based approach with a classical ML schema to leverage local context. Meanwhile, Li et al. (2018) tackled the dynamic typing issue by employing Deep Learning (DL), specifically a Recurrent Neural Network (RNN) coupled with a Pointer Network.

In this chapter, we quantitatively compare these two approaches using a large corpus of GitHub Python files. We also introduce an ensemble approach named *Extended Network*, where a neural network determines the schema used for each prediction. The exemplified *Extended Network* model slightly outperforms its individual components. Our comprehensive evaluation and analysis demonstrate the potential of ensemble-like methods for code completion and recommendation in dynamically typed languages.

Section 4.1 discusses our motivation in this study, while Section 4.2 provides a summary of background knowledge and related work. The *Extended Network* architecture and its model are described in Section 4.3. We evaluate the proposed approach and answer to the second core research question (CRQ2) in Section 4.4. Finally, the chapter is concluded by Section 4.5.

This chapter is based on our peer-reviewed publication (Schumacher et al., 2020).

4.1 Introduction

Code recommendation systems for software engineering aim to speed up the development of large software projects (Robillard et al., 2010). A specific type of these systems is exhibited by the well-known features in IDEs: code completion and next token prediction (Allamanis et al., 2018). In large software projects, these features are particularly useful for identifying the applicable function from numerous options, navigating code, and learning new libraries, as they provide information on available functions, methods, or attributes (Schumacher, 2019).

However, these code recommendation systems can only suggest an approximation of user's next inputs in a text editor. This is attributed to the necessity of simplifying assumptions about human behavior and the vast number of possible inputs, which makes accurate code prediction difficult (Schumacher, 2019). Furthermore, unlike text in Natural Language Processing (NLP), a code recommender must infer both value and object type of the code token that the user intends to define next.

Challenges of dynamically typed languages. Meanwhile, multiple models proposed for code completion struggle to predict the specific value of a code token, whereas merely identifying the token type is less challenging (Liu et al., 2017; Raychev et al., 2016; Li et al., 2018; Bielik et al., 2016). In these studies, the syntax of a programming language can enhance prediction accuracy by narrowing down the search space of possible token types (Schumacher, 2019). ML techniques can be trained to understand this syntax, aiding in the prediction of the next token type, i.e. non-terminal nodes in Abstract Syntax Trees (ASTs).

Nevertheless, most intelligent code completion approaches depend on type information, that is only available at compile time, limiting their applicability to dynamically typed languages such as JavaScript or Python (Liu et al., 2017). As a result, providing code completion in these languages is challenging due to the absence of type information during programming. With dynamically typed languages gaining popularity, there is an increasing need for intelligent code completion tailored to these languages (Schumacher, 2019).

Notable approaches. Additionally, the emergence of large code repositories, often referred to as *Big Code* (e.g. GitHub), has spurred the adoption of DL and probabilistic language models for code analysis (Allamanis et al., 2018). Although RNNs have been widely used in predicting subsequent code tokens, they face certain problems. Notably, the computational complexity of evaluating the `softmax` function utilized in RNNs can become significant, escalating with the size of the global vocabulary (Li et al., 2018; Schumacher, 2019).

One strategy frequently employed to tackle this issue involves constraining the range of the vocabulary used. However, this method gives rise to another obstacle: Out-of-Vocabulary (OOV) words that are unforeseeable by RNNs (Li et al., 2018). To address this matter, Li et al. (2018) proposed the Pointer Mixture Network model, which utilizes Pointer Networks (Vinyals et al., 2015) to predict OOV words. Nonetheless, even with this model, the ability to predict OOV words remains limited to the current context (Schumacher, 2019).

Another promising direction for code recommendation challenges is adapting statistical models like Probabilistic Higher Order Grammar (PHOG), as presented by Bielik et al. (2016). This model incorporates production rules derived from a context-sensitive grammar, offering flexibility in accommodating varied vocabulary sizes and addressing long-range dependencies.

Contributions of this work. Given the advantages of the aforementioned methods and the established efficacy of ensemble learning (Sagi et al., 2018), we proposed an ensemble-like architecture that integrates neural and classical ML approaches to mitigate the weaknesses of individual models. To demonstrate the usability of our proposed architecture, we combined a Pointer Mixture Network and PHOG to predict terminal values of AST nodes. Our contributions in this work are as follows:

- *An Extended Network architecture* incorporating neural and classical ML models with a selection mechanism. The architecture is exemplified through a combination of Pointer Mixture Network and PHOG for predicting next code tokens. Furthermore, we featured an enhanced RNN architecture with multiple Long Short-Term Memory (LSTM) layers and dropout. This model was implemented by Schumacher (2019) in his Bachelor's thesis conducted within our research group.

- *An extensive comparison* to evaluate the accuracy of the proposed model across various settings of relevant parameters. In addition, the performance of individual components is examined to ascertain their respective strengths.

Notably, our work in this chapter employs code completion approaches to primarily address the *programming barrier* </> (discussed in Chapter 1, Section 1.1), serving as our answer to the second core research question (CRQ2), specified in Section 1.4 of Chapter 1.

4.2 Background and Related Work

As detailed in Chapter 2, Section 2.2, the advancement of ML models for code is driven by the *naturalness hypothesis*, which views code completion through a natural language lens. Besides, NLP can be defined as the convergence of Artifical Intelligence (AI) and linguistics (Nadkarni et al., 2011; Schumacher, 2019). This section delineates background information and related work, laying the groundwork for our proposed model in the subsequent section.

4.2.1 The Naturalness Hypothesis

According to Allamanis et al. (2018), software can be considered as a form of interaction akin to human language. This hypothesis enables the application of software corpora in a similar manner to Natural Language (NL) corpora. Extensive research has been conducted to further investigate this resemblance, revealing the predictability of code. Notably, studies utilizing *n-grams* (Hindle et al., 2016; Tu et al., 2014) have highlighted the naturalness and repetitiveness of code (more details in Chapter 2, Section 2.2.2). Additionally, various ML and DL models rooted in NLP have shown effectiveness in handling code (Liu et al., 2017; Raychev et al., 2016; Li et al., 2018; Mikolov et al., 2013c).

4.2.2 Typical Machine Learning Models for Code Completion

In the context of code completion, ML techniques are applied by interpreting code tokens as NL and representing program code using ASTs. The task of code completion is then formulated as predicting the subsequent node in the AST given a certain input or a part of the AST (Schumacher, 2019). Various typical neural network architectures such as RNN, LSTM and word embedding technique (Mikolov et al., 2013c; Mikolov et al., 2013a) have been deployed to tackle this problem. Moreover, the efficacy of these neural networks is notably enhanced as the volume of available data for training and testing increases (Allamanis et al., 2018).

Leveraging this benefit of large corpora, Liu et al. (2017) presented an LSTM architecture designed to acquire code completions by analyzing a substantial dataset of dynamically typed JavaScript code. Their model stands as a competitor to existing probabilistic code models, such as decision trees (Raychev et al., 2016; Schumacher, 2019). Nevertheless, conventional neural approaches encounter challenges related to capturing long-range dependencies and dealing with the OOV issue (Li et al., 2018). Further details of neural network architectures for code completion are discussed in Section 2.2.4 of Chapter 2.

4.2.3 PHOG and Pointer Mixture Network

Among various ML and DL approaches explored for code completion (outlined in Sections 2.2.3 and 2.2.4 of Chapter 2), Probabilistic Higher Order Grammar (PHOG) introduced by Bielik et al. (2016) and Pointer Mixture Network proposed by Li et al. (2018) are particularly suitable for our ensemble-like architecture.

PHOG draws inspiration from its predecessor models such as Probabilistic Context Free Grammars (PCFG) and *n-gram* (Jurafsky et al., 2008; Gvero et al., 2015; Hindle et al., 2016). In particular, PHOG extends the capabilities of PCFG by conditioning not only on the parent non-terminal node but also by dynamically constructing contextual information through traversal of the AST (Bielik et al., 2016; Schumacher, 2019).

Contrary to PCFG and *n-gram* models, PHOG adaptively generates program representations based on the obtained contextual information. This flexible program representation makes PHOG applicable to any programming language featuring ASTs. As a result, while PCFG and *n-gram* models are oriented toward specific domains, limiting their generalizability, PHOG offers broader applicability (Schumacher, 2019). More insights on these models are provided in Section 2.2.3 of Chapter 2.

While PHOG represents classical ML methods for code, **Pointer Mixture Network** adopts DL techniques, particularly LSTM architecture and neural attention, for code completion. Specifically, Pointer Mixture Network integrates an attentional-LSTM with a Pointer Network to address long-range dependencies and OOV words (Li et al., 2018).

The general concept of attentional-LSTM targets to mitigate the limitations posed by storing information within a single fixed-length vector in RNNs (i.e. the *hidden state bottleneck*) and to effectively capture long-range dependencies. The attention mechanism (Bahdanau et al., 2015) was proposed for this purpose, employing a weighted sum of both past and current hidden state vectors. This mechanism enables information propagation across extended time intervals.

Particularly, the attentional-LSTM of Li et al. (2018) comprises two forms of attention: *context attention* and *parent attention*. The *context attention* mechanism operates within a fixed-sized context window, while the *parent attention* is a distinct type of *context attention* tailored for ASTs. The latter form incorporates the hidden state from the parent node of the currently processed node into the attentional layer (Schumacher, 2019).

Subsequently, based on the *localness of software* premise (Tu et al., 2014), which suggests that source code tends to repeat locally, Li et al. (2018) adapted Pointer Networks (Vinyals et al., 2015) to handle the issue of static output vocabularies. Essentially, a Pointer Network functions

by *pointing* to a token within the input sequence to predict the next token. Further details on the Pointer Mixture Network and its groundwork are clarified in Chapter 2, Section 2.2.4.

4.3 Extended Network

This section presents our *Extended Network* approach and its exemplified model, which was developed by Schumacher (2019) through his Bachelor study conducted in our research group.

4.3.1 The Core Idea

Our *Extended Network* aims to enhance code modeling and prediction by combining neural and probabilistic language models into an ensemble-like architecture. In addition, we employed a dynamic selection mechanism to identify suitable components used for each prediction based on their capabilities. By this way, deficiencies in one component are balanced out by the strengths of others, fostering more accurate predictions.

Schumacher (2019) established two prerequisites for constructing an instance of the *Extended Network*: a foundational neural network model and clear criteria for component selection. The ensemble model's output layer includes the neural network's output-dimensions, representing words from the vocabulary, and additional output-dimensions, one for each probabilistic model. These extra dimensions convey the estimated probability of the corresponding probabilistic language model generating a fitting prediction for the provided input.

Formal Definition

To formally define the output of the *Extended Network*, let y signify the output layer of a given neural network:

$$y = f(Wx + b) \tag{4.1}$$

with f being an activation function, while $W \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$ are trainable parameters (Schumacher, 2019). The output of the neural network $y \in \mathbb{R}^n$ is a probability distribution over n potential outputs, while $x \in \mathbb{R}^m$ denotes m-dimensional embedding of an input code token.

The *Extended Network* incorporates a set of probabilistic language models known as g_i (where i ranges from 1 to k) into the above neural network. This integration expands the output-dimensions of the neural network, yielding an extended output layer, referred to as $y' \in \mathbb{R}^{n+k}$. To obtain y', we adjusted the dimensions of parameters W and b. Subsequently, the output of the *Extended Network* is then computed as follows:

$$y' = f(W'x + b') \tag{4.2}$$

where $W' \in \mathbb{R}^{(n+k) \times m}$ and $b' \in \mathbb{R}^{n+k}$ are trainable parameters and the activation function f remains unchanged.

Besides, instead of solely providing a probability distribution across the predefined vocabulary, the *Extended Network* additionally assigns a probability to each probabilistic language model, indicating the likelihood of model g_i ($i \in [1, ..., k]$) producing accurate predictions. We labeled

the outcomes $y'_{n+1}, ..., y'_{n+k}$ generated by probabilistic language models as `special_IDs`. These outputs individually function as distinct identifiers for their respective probabilistic language models (Schumacher, 2019).

Component Selection Procedure

Given an instance of the *Extended Network* architecture encompassing a neural network and k single probabilistic or statistical language models, the output can be either a vocabulary word or a `special_ID` from a statistical model. Figure 4.1 depicts the model selection mechanism in this case. Namely, if the value of $argmax(y')$ from the computed probability distribution y' aligns with a `special_ID`, the output is derived from the relevant statistical model; otherwise, it is determined by $argmax(y')$ as usual.

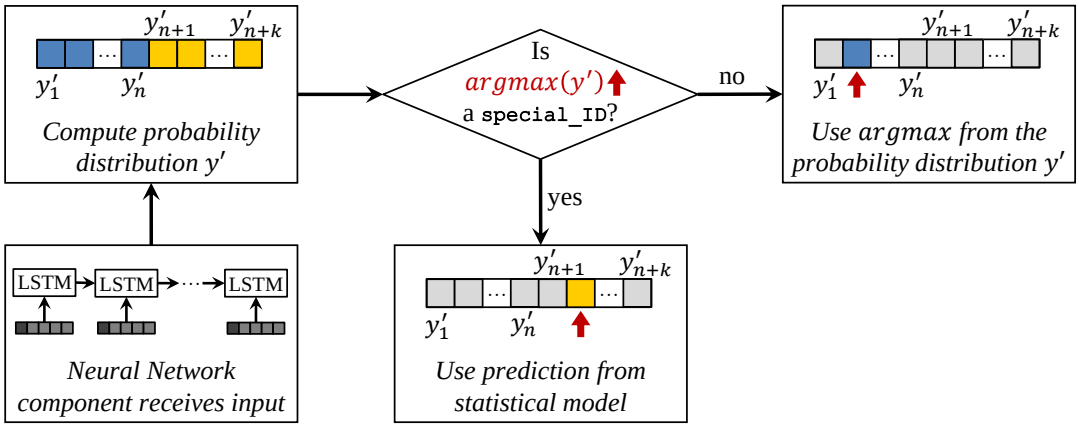


Figure 4.1: Flowchart displaying the component selection in the *Extended Network* during prediction, adapted from Figure 3.4 of Schumacher (2019).

The process of learning when to engage the neural network component or alternative ones within the *Extended Network* involves creating unique deterministic labels. These labels are formed from a selection of either vocabulary words or designated `special_IDs`. Schumacher (2019) developed a hierarchical structure wherein a series of conditions $C = \{c_0, ..., c_k\}$ is sequentially evaluated to identify these labels.

Particularly, condition c_0 is dedicated to the neural network component, while condition c_i is associated with probabilistic language models g_i for $i \in [1, ..., k]$. Upon fulfilling a condition, the matching label of this condition is assigned to the related `special_ID` or word. This hierarchical method facilitates the generation of deterministic labels while giving precedence to individual components based on the order in which conditions C are assessed.

Consequently, the composition and sequence of conditions C play a crucial role in shaping the performance of the *Extended Network*. However, there is no universal approach to formulate these conditions, as their effectiveness is closely tied to the specific application of the *Extended Network* (Schumacher, 2019). The following subsections demonstrate the utility of our proposed ensemble-like architecture using a model that comprises Pointer Mixture Network as the neural network component and PHOG as the statistical model.

4.3.2 An Illustrative Model for the Extended Network Architecture

To exemplify our proposed architecture, we designed an ensemble model for recommending next code tokens (i.e. terminal values of AST nodes) in Python programs. This model targets to mitigate the OOV occurrences that are unpredictable with long-range dependencies. As discussed in Section 4.2.3, we utilized Pointer Mixture Network (Li et al., 2018) and PHOG (Bielik et al., 2016) as components of the model. Hereafter, the phrase "*Extended Network*" denotes the proposed concept, while "*Extended Network model*" refers to the illustrative model.

Improving Pointer Mixture Network with PHOG

Essentially, the Pointer Mixture Network consists of an RNN, implemented as an attentional-LSTM, and a Pointer Network. When the RNN component fails to forecast OOV words, the prediction is typically handled by the pointer component. However, this approach by Li et al. (2018) experiences obstacles with long-range dependencies.

These challenges stem from the authors' utilization of a fixed-size *attention window* of previous hidden states for both components, as detailed in Section 2.2.4 of Chapter 2. This configuration confines the RNN to attending only to preceding information within the window and restricts the pointer component to referencing input words within the same range. Consequently, neither component can predict OOV words beyond the *attention window* (Schumacher, 2019).

We addressed this issue by adopting the PHOG model. One of its key advantages is that it establishes the vocabulary space based solely on the unique words presented in the training data. This characteristic enables the model to be unrestricted by vocabulary size and reduces challenges associated with long-range dependencies (Schumacher, 2019).

Additionally, PHOG incorporates TCOND, a Domain-Specific Language (DSL) tailored for navigating AST structures and collecting information during traversal (outlined in Section 2.2.3 of Chapter 2). By leveraging TCOND, the model can gather contextual information without being constrained by word positions within the AST, hence alleviating the problem posed by the *attention window*.

Prediction Strategy

As delineated in the component selection procedure (Section 4.3.1), output of the *Extended Network* can be a vocabulary word or a `special_ID`. The latter reveals the necessity of inspecting suggestions from the relevant probabilistic language model. In our *Extended Network model*, outputs are derived from either Pointer Mixture Network or PHOG. Therefore, we labeled the `special_ID` in this context as `hog_ID`.

More precisely, the *Extended Network model* offers three output options: (i) from the RNN component, (ii) from the Pointer Network, and (iii) from PHOG. The model learns to choose the appropriate option for each prediction. To facilitate this process, an additional output-dimension for the `hog_ID` was incorporated into the output layer of the RNN component. A set of conditions was then established to guide this learning process. Given the significance of the condition set, we dedicate the following subsection to explain the conditioning process in detail.

4.3.3 Component Selection in the Extended Network Model

Inspired by the Pointer Mixture Network, we trained the *Extended Network model* to select suitable component for each prediction. Particularly, every next node value in an AST is assigned a deterministic label, specifying the component used for predicting this node value. These labels are obtained by assessing individual values against a set of predefined conditions.

Furthermore, we compiled these deterministic labels into a file, forming a *terminal corpus* (named `tcorpus`), where each line represents labels for one AST in the dataset. This corpus serves as the ground truth for the network during the learning process. Detailed explanations regarding the condition set and the creation of the *terminal corpus* are provided below.

Conditioning the Extended Network Model

As outlined in Section 4.3.2, the *Extended Network model* encompasses an RNN, a Pointer Network, and a PHOG component, attaining output words from the vocabulary, the *attention window*, and PHOG's prediction, respectively. Schumacher (2019) defined a set of conditions in hierarchical order to determine the activation of each component for predicting an AST node.

Overall strategy. Fundamentally, if the node value corresponds to an entry in the terminal dictionary, which is a dictionary of vocabulary word and ID pairs, then the RNN component is utilized. Otherwise, if the value resides within the *attention window*, the Pointer Network component is chosen. Alternatively, recommendations from the PHOG component are considered. However, if none of the components successfully predict the value, the node value is treated as unknown, denoted by `unk_ID`.

Encoding labels. In particular, we annotated the terminal dictionary as `tdict`, where `tdict[value]` returns the matching ID of the terminal value (i.e. ID of the vocabulary word). Consequently, labels for node values are encoded using (i) `tdict[value]`, (ii) location indices from the *attention window*, (iii) `hog_ID`, or (iv) `unk_ID`, implying the usage of RNN, *attention window*, PHOG, or neither of the components, respectively. Figure 4.2 illustrates the procedure applied to terminal `T` with value `foo` and type `Str`.

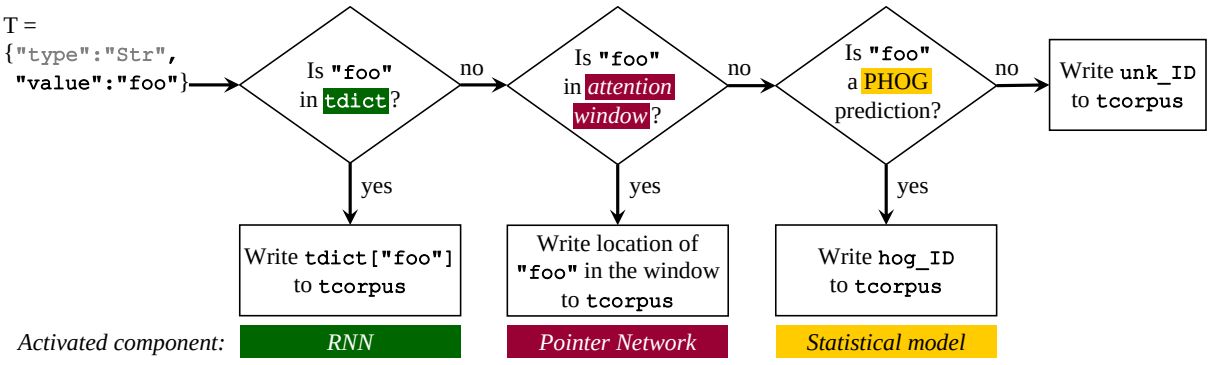


Figure 4.2: An example of generating a label for terminal `T` and writing it to the terminal corpus `tcorpus`, adapted from Figure 3.6 of Schumacher (2019).

Namely, if the node value `foo` is in the terminal dictionary `tdict`, its ID (i.e. `tdict["foo"]`) is appended to the current line of the *terminal corpus* (`tcorpus`), indicating the utilization of

the RNN component. Conversely, if the value `foo` is absent from `tdict` but is located in the *attention window*, its position within the window is noted in the ongoing line of `tcorpus`. This scenario signifies that the Pointer Network component is selected for prediction.

However, if neither of the preceding cases is valid, then PHOG's suggestion is evaluated. If the output from PHOG aligns with the node value, the `hog_ID` is added to `tcorpus`, confirming PHOG as the chosen predictor. Finally, when none of the components yield the considered value (`foo`), an `unk_ID` is assigned as the label of the node value.

Creating Terminal Corpus

The encoding process described above is applied to each node within an AST and across all ASTs in the dataset, forming the *terminal corpus* `tcorpus`. Algorithm 4.1, developed by Schumacher (2019) and based on the approach of Li et al. (2018), displays this construction. The `PROCESS` function takes as inputs a file of ASTs `asts_file`, a PHOG's predictions `hog_file`, a terminal dictionary `tdict` containing the n most frequent words (with n as the vocabulary size), and the *attention window* size `attn_size`.

Algorithm 4.1 Creating the terminal corpus, adapted from Algorithm 1 of Schumacher (2019)

Input: *ASTs* (*asts_file*), *PHOG's predictions* (*hog_file*), *terminal dictionary* (*tdict*), *attention window size* (*attn_size*)
Output: *terminal corpus*

```
1  function PROCESS(asts_file, hog_file, tdict, attn_size)
2      tcorpus ← []
3      for ast, hog_pred in asts_file, hog_file do
4          tline ← []
5          attn_queue ← DEQUE(attn_size)
6          for node, hog_node in ast, hog_pred do
7              if "value" in node.keys then
8                  dict_value ← node["value"]
9                  if dict_value in tdict then
10                     tline.append(tdict[dict_value])
11                     attn_queue.append("NormaL")
12                 else
13                     if dict_value in attn_queue then
14                         attn_loc ← GET_LOC(attn_queue, dict_value)
15                         tline.append(attn_loc)
16                     else if dict_value == hog_node["value"] then
17                         tline.append(hog_ID)
18                     else
19                         tline.append(unk_ID)
20                     attn_queue.append(dict_value)
21             else
22                 attn_queue.append("EmptY")
23                 tline.append(tdict["EmptY"])
24         tcorpus.append(tline)
25     return tcorpus
```

Here, the `hog_file` is a JavaScript Object Notation (JSON) file, collecting predictions made by a trained PHOG for every AST in the training and testing datasets. The structure of `hog_file` mirrors how AST nodes are organized in these datasets. In particular, each line in this file corresponds to an AST and is formatted as a separate JSON object. Within each object, recommendations for value and type are provided for every node present in the AST.

Overall, for each AST, we create a line called `tline`, where the value of each terminal node is converted to an identifier from the terminal dictionary `tdict` (lines 9–11), an index denoting its position within the *attention window* (lines 13–15), or the `hog_ID` (lines 16–17). If the *Extended Network model* is unable to predict the current terminal value, `unk_ID` is appended to the `tline` (line 19). The final output, `tcorpus`, is a collection of lines, with each line pertaining to the terminal labels of an individual AST.

In our implementation, each label is encoded by an unique integer. Additionally, non-terminal nodes without value fields have their values designated as `EmptY` (lines 22–23). Furthermore, we employed `attn_queue` as a deque object (line 5), with a maximum length bound to `attn_size`, to store values in the *attention window*. Entries in this deque comprise `NormaL` for terminal values in `tdict`, the terminal value itself, and `EmptY` for non-terminal nodes. The unconventional capitalization prevents conflicts with other node values.

Predicting Unknown Words

We designed the *Extended Network model* to consistently generate a prediction by excluding `unk_ID` as a valid choice. As a result, any predictions of `unk_ID` are deemed incorrect, prompting the model to steer clear of such predictions. While it is technically possible to choose the PHOG component each time an `unk_ID` label appears, irrespective of its accuracy, this method could overly bias the *Extended Network model* towards favoring predictions of `hog_ID`.

Specifically, our model prioritizes the PHOG component when its prediction is accurate rather than uncertain. In addition, even though the RNN component may not always yield the correct outcome, it can produce results closely resembling the correct one. In contrast, due to the lack of a similarity metric for PHOG's predictions, assessing the proximity of an incorrect PHOG prediction to the true label is challenging. Consequently, when PHOG fails to offer a correct answer, the output of the RNN component takes precedence (Schumacher, 2019).

4.4 Evaluation

This section details our experimental setup and results, followed by our response to the second core research question (CRQ2), defined in Chapter 1, Section 1.4.

4.4.1 Experimental Setup

We leveraged the Python150k dataset provided by ETH-Zürich[1] in our experiments due to its widespread use in code completion context (Bielik et al., 2016; Liu et al., 2017; Li et al., 2018).

[1]SRILAB, `https://www.sri.inf.ethz.ch/py150`, (Accessed: 13 March 2024).

This dataset contains 150,000 Python scripts from GitHub repositories. To ensure uniqueness, duplicate entries, including forks, were eliminated. Moreover, only repositories with permissive and non-viral licenses, such as MIT, Apache, or BSD were included. Besides, each Python script was parsed into AST format utilizing the parser in Python 2.7 (Raychev et al., 2016).

Dataset Splitting

Initially, the whole dataset was divided into 100,000 ASTs for training purposes and 50,000 ASTs for testing. Subsequently, for parameter tuning and model development, we continued partitioning the training dataset into training and development subsets. In particular, from the 100,000 training ASTs, 90,000 were designated for training and 10,000 for development, with the evaluation set (50,000 ASTs) left unchanged.

Applying this specific training-development split for parameter tuning helps maintain the integrity of the testing dataset and enables compatible comparisons with other models trained on the Python150k dataset. This approach also allows us to measure accuracy on previously unseen data since the testing dataset was reserved solely for the final evaluation of our model, and no parameters were adjusted using this testing dataset (Schumacher, 2019).

Data Preprocessing

In the *Extended Network model*, target values are derived by shifting the input sequence (i.e. flattened AST) forward one time step. For instance, given an input sequence $in_0 ... in_t$ at time step t, the target sequence is $in_1 ... in_{t+1}$. Furthermore, as outlined in Section 4.3.3, we constructed a terminal corpus of encoded labels to condition the model on selecting the appropriate component for every prediction task. This corpus was created for both train-dev and train-test splits, accommodating vocabulary sizes of 1,000 and 10,000 words. Table 4.1 shows the rules for assigning unique integers to each label.

Table 4.1: Encoding labels in the terminal corpus of the *Extended Network model*.

Label	**Integer value**
`EmptY`	0
`tdict_start_idx`	1
`tdict_end_idx`	`tdict_size` − 1
`unk_ID`	`tdict_size`
`hog_ID`	`tdict_size` + 1
`eof_ID`	`tdict_size` + 2
`attn_start_idx`	`tdict_size` + 3
`attn_end_idx`	`attn_start_idx` + `attn_size`

Particularly, the value 0 is reserved for the label `EmptY`, while integers from 1 to `tdict_size`−1 are allocated to the terminal dictionary `tdict`, where `tdict_size` is the dictionary size. The values for `unk_ID` and `hog_ID` follow the last index of the terminal dictionary `tdict_end_idx`. The end-of-file identifier `eof_ID`, used as padding, is calculated next. Finally, indices within

the *attention window* are computed subsequently and constrained to the range of the attention window size `attn_size`, as described in the last two rows of Table 4.1.

Experiment Configuration

Our *Extended Network model* integrates Pointer Mixture Network and PHOG. Therefore, we adapted hyperparameters from Li et al. (2018) for the former component. For the latter, the PHOG model developed by Bielik et al. (2016) was employed and trained on the training dataset (either from the train-dev or train-test split). Table 4.2 summarizes our main configuration.

Table 4.2: Experiment configuration for the *Extended Network model*.

Parameter	**Value**
LSTM unrolling length	50
Attention window size	50
Hidden unit size	1,500
Optimizer	Adam
Number of epochs	8

In particular, the LSTM within the Pointer Mixture Network of our *Extended Network model* was configured with specific hyperparameters: an unrolling length of 50, an *attention window* of size 50, and a hidden unit size of 1,500. Besides, to optimize the model during training, we utilized the cross entropy loss function along with stochastic gradient descent and the `Adam` optimizer (Kingma et al., 2015).

Additionally, to address potential gradient issues, a clipping threshold of 5 was applied to prevent gradients from exploding, as recommended by Li et al. (2018). Training sessions were capped at a maximum of 8 epochs, starting with a learning rate of 0.001 and applying a decay rate of 0.6 after each epoch (Schumacher, 2019).

Evaluation Strategy

We performed three main sets of experiments: (i) training a single-layer *Extended Network model* without dropout, (ii) increasing the number of layers to two and adjusting dropout rates, and (iii) utilizing the train-test split to train the most promising models from (i) and (ii) on vocabulary sizes of $1k$ and $10k$. Evaluation metrics were the same among all experiments. Notably, for the second experiment, we measured accuracies on the training and development datasets and observed the differences between both sets.

In addition, we conducted two further analyses. The first one compares the performance of individual components within the proposed model, while the second one investigates the impact of AST nodes without values on each component's performance. These analyses examined four different aspects of every component: (1) predictive ability, (2) selection likelihood as a predictor, (3) accuracy when chosen, and (4) independent performance. Calculation details and results are elaborated in the next subsection.

4.4.2 Experimental Results

This subsection reveals our evaluation results, beginning with the main experiments and followed by the aforementioned two analyses.

Evaluation Results for the Three Sets of Experiments

> *Finding 1.* The *Extended Network model* slightly outperforms its components and can be further enhanced by adding an extra LSTM layer with a 20% dropout rate.

Single layer Extended Network model. In our first experiment, we omitted the use of dropout and multi-layers to exclusively evaluate the influence of the *Extended Network model* on accuracy. This approach ensures a fair comparison with the Pointer Mixture Network, as multi-layers and dropout are likely to enhance its performance as well, thereby potentially diminishing the benefit gained from implementing an *Extended Network model* (Schumacher, 2019).

After training the Pointer Mixture Network for 7 epochs on the designated train-test split, we successfully replicated the findings of Li et al. (2018). At this point, the *Extended Network model* exhibited an accuracy enhancement of 0.6% on the test-set and 1.9% on the train-set, outperforming the Pointer Mixture Network. Specifically, Table 4.3 demonstrates the improved performance of the *Extended Network model* in predicting next node values, surpassing both the Pointer Mixture Network and PHOG models.

Table 4.3: Accuracy on the test-set of a single layer *Extended Network model*, without dropout, trained for 7 epochs on train-test split, vocabulary size of $1k$.

Model	Accuracy on test dataset
PHOG	63.8%
Pointer Mixture Network	66.4%
Extended Network model (ours)	**67.0%**

Two-layer Extended Network model with dropout rates. The above experiment also highlighted a notable difference in accuracy between the training and testing datasets, showing a margin of over 4%. This disparity prompted us to adopt the dropout technique for the underlying LSTM network, inspired by the work of Gal et al. (2016). The objective was to enhance the generalization capability of the *Extended Network model* and thereby improve its accuracy on the test dataset (Schumacher, 2019).

To determine the optimal dropout rate, we employed the train-dev split and configurations outlined in Table 4.4. Moreover, we augmented the network depth by introducing a second layer to the LSTM. As a result, incorporating a dropout rate of 20% and an additional layer achieved the highest development accuracy (67.0%) and reduced the training-development accuracy gap from 3.3% to 1.5%. Notably, the training accuracy remained consistent at the 20% dropout rate, regardless of whether a single- or two-layer LSTM was applied, indicating that adding the second layer did not cause over-fitting.

Table 4.4: Accuracies for a single layer *Extended Network model* and a two-layer model across different values of dropout, vocabulary size of $1k$.

# of layers	Dropout percentage	Dev-set accuracy	Train-set accuracy
Single layer	0 %	66.6 %	69.9%
	20 %	66.7 %	68.5%
	40 %	66.3 %	67.6%
Two layers	15 %	66.7 %	68.1%
	20 %	**67.0 %**	68.5%
	30 %	66.5 %	67.6%

Two-layer Extended Network model (with dropout) on two vocabulary sizes. In our final experiment, we utilized the two-layer *Extended Network model* with a 20% dropout rate and the original train-test split of the Python150k dataset. Table 4.5 presents the outcomes of the *Extended Network model* for vocabulary sizes of 1,000 and 10,000 words. In this experiment, we compared accuracies of our model with probabilistic language models like PHOG (Bielik et al., 2016) and decision trees (Raychev et al., 2016), alongside the neural model of Pointer Mixture Network (Li et al., 2018).

The results illustrated that under the current settings, our *Extended Network model* reached peak performance with a $10k$ vocabulary, slightly outperforming the decision tree method and two components of the model by margins ranging from 0.1% to 5.5%. It is worth mentioning that training the *Extended Network model* on an NVIDIA Tesla V100 GPU required approximately 1.5 hours per epoch with a $1k$ vocabulary and 8 hours per epoch with a $10k$ vocabulary.

Table 4.5: Accuracy of the two-layer *Extended Network model* (20% dropout), compared to state-of-the-art probabilistic language models and the original Pointer Mixture Network.

Model	Accuracy on test dataset	
PHOG	63.8%	
Decision Trees	69.2%	
	$1k$ vocab	$10k$ vocab
Pointer Mixture Network	66.4%	68.9%
Extended Network model (ours)	67.5%	**69.3%**

Component Comparison

To understand the impact of individual components in the implemented *Extended Network model*, we conducted a supplementary analysis using the same model configuration as prior experiments. The performance of each component was evaluated throughout several aspects. Besides, we examined recommended values of these components on the evaluation dataset (i.e. 50,000 ASTs) with a vocabulary size of $10k$. Furthermore, our investigation particularly targets value prediction, namely, forecasting terminal values within AST nodes.

> *Finding 2.* RNN is the predominant component in the *Extended Network model*, although this outcome could be affected by the presence of `EmptY` tokens in the training and testing datasets, as clarified in the subsequent analysis.

Table 4.6 displays the accuracy of each component within the *Extended Network model*. The final three columns correspond to the RNN (i.e. LSTM-based), Pointer Network (i.e. attention mechanism), and PHOG components integrated into the model. Meanwhile, the first column comprises four distinct criteria assessed on every component.

Table 4.6: Accuracies of components in the *Extended Network model**, including predictions for nodes without values (i.e. labeled as `EmptY`).

Aspect	RNN	Pointer Network	PHOG
Able to predict	84.2%	5.4%	100%
Used as a predictor	91.5%	1.1%	7.4%
Used and correct	54.1%	0.2%	0.4%
Correct own predictions	59.1%	20.2%	5.4%

* Vocabulary size is 10,000. These components were evaluated on a test-set of 50,000 ASTs.

Firstly, the "**Able to predict**" row indicates the proportion of instances where a component could potentially make predictions. For instance, if the target token is absent from the terminal dictionary `tdict`, the RNN cannot make any predictions. It is worth noting that PHOG is unaffected by these restrictions and can always make predictions, hence 100% in this case.

The "**Used as predictor**" row signifies the utilization rate of each component for predictions. It also portrays the distribution of the component selection procedure, illustrated in Figure 4.1. Generally, RNN is the most frequently used component, with the others selected roughly in accordance with their accuracy. The prevalence of the RNN component might be attributed to its specialization in predicting `EmptY` tokens, which are frequently encountered in the dataset. Details of this matter are discussed through our second analysis below.

Meanwhile, the "**Used and correct**" row shows the percentage of correct predictions made by each component when selected. Overall, the RNN component demonstrates the highest accuracy in prediction, whereas the pure Pointer Network and PHOG have minimal impact.

Ultimately, the "**Correct own predictions**" row reveals how accurately each component performs when used individually. The percentages represent the proportion of correct predictions made by each component relative to its total predictions. Notably, the RNN component achieves an accuracy of nearly 60% for its own predictions. Nevertheless, this figure might be misleading due to the prediction of `EmptY` token (explained below). Conversely, PHOG exhibits the lowest precision, with approximately 5% of correct predictions.

Impact of AST Nodes without Values

As previously outlined, our second analysis inspects the influence of AST nodes without values on the performance of components within the *Extended Network model*.

> *Finding 3.* The prediction accuracy of the RNN component decreases markedly without considering `EmptY` nodes in the testing dataset.

`EmptY` nodes. A large portion of nodes within ASTs in the dataset, and generally in typical Python source code, lack assigned values. These nodes often correspond to non-terminal nodes or terminal nodes characterized by fixed keywords (e.g. `as`). Following Li et al. (2018), we treated all nodes exhibiting this property as a special value in the terminal dictionary `tdict`. This special case is denoted as `EmptY` in Section 4.3.3, or as `EMPTY` in Figure 4 of Li et al. (2018).

Predicting `EmptY` nodes is exclusively allocated to the RNN component. Essentially, value suggestions for these nodes should entail subsequent invocation of a separate prediction model to determine the *type* of these AST nodes and, if applicable, propose a unique representation for completing the node in the source code, as discussed by Li et al. (2018).

Issues arising from `EmptY` nodes. However, training and evaluating models using the value `EmptY` presents challenges. Primarily, a significant portion of AST nodes (47.6% of over 29,9 million predictions) are `EmptY` nodes, potentially biasing the RNN model towards this specific value during training. Besides, predicting this value might be comparatively easier than others, as the context of AST nodes without values might provide hints of "not present" for other node values. For example, encountering the keyword `as` shortly after `with` is quite probable.

In general, including the predictions of this special value in the evaluation significantly boosts the accuracy values, which we also observed in our experiments. In our main evaluation, we followed the approach of prior studies (Li et al., 2018; Bielik et al., 2016) and included predictions of the `EmptY` nodes in order to provide comparable results. However, we did not consider such an evaluation a realistic one, since it might not correspond to a perceived user experience. In fact, other state-of-the-art works (Brockschmidt et al., 2019; Alon et al., 2020a) report much lower accuracy values for code completion, e.g. accuracy@5 of 24.83% for Java (Alon et al., 2020a).

Additional outcomes without `EmptY` nodes. Consequently, we supplied complementary results of an evaluation with all AST nodes, but omitting the special value `EmptY`. It is worth mentioning that such nodes are still used in the training, and so the presented results might be worse than in a setting where they are not considered at all. Table 4.7 displays the results in this scenario, with column and row names having the same meaning as those in Table 4.6 above.

Table 4.7: Accuracies of components in the *Extended Network model**, excluding predictions for nodes without values (i.e. labeled as `EmptY`).

Aspect	RNN	Pointer Network	PHOG
Able to predict	70.0%	10.3%	100%
Used as a predictor	84.3%	2.0%	13.7%
Used and correct	18.8%	0.4%	0.7%
Correct own predictions	22.3%	20.7%	5.6%

* Vocabulary size is 10,000. These components were evaluated on a test-set of 50,000 ASTs.

The accuracy of the RNN component declines significantly when selected, from 54.1% to 18.8%. This indicates that the higher accuracy reported for this component in Table 4.6 is

primarily due to correct predictions of `EmptY` values. Conversely, the individual performances of the PHOG model and the Pointer Network remain largely unmodified.

Despite the reduced accuracy of the RNN component, it is still the preferred choice, with an invocation rate of 84.3%, likely due to the training process with `EmptY` values. This dominance significantly affects the overall performance of the *Extended Network model*, decreasing it from 54.7% (with `EmptY` values) to 20.0% (without `EmptY` values) in the "Used and correct" case.

4.4.3 Response to CRQ2

Our second core research question (CRQ2), outlined in Chapter 1, Section 1.4, investigates whether ensembles of ML-based predictors enhance code completion accuracy. Even though the exemplified *Extended Network model* only modestly outperforms its individual components, i.e. Pointer Mixture Network and PHOG (refer to *Finding 1*, Tables 4.3 and 4.5), the results still underscores the advantage of an ensemble model over its constituents, hence addressing the posed core research question.

> **A-CRQ2. Ensembles of ML-based code completion models improve accuracy for individual components, with overall performance impacted by ensemble size, model selection mechanism, and underlying neural network architecture.**

4.5 Summary

This chapter presents an ensemble approach for predicting next tokens in dynamically typed languages, directly tackling the *programming barrier* </> (defined in Chapter 1, Section 1.1). For illustration, we proposed an *Extended Network model*, encompassing Pointer Mixture Network and PHOG components.

The evaluation results demonstrate the improved accuracy of the proposed model, compared to its standalone components. These results also serve as a response to the second core research question (CRQ2), specified in Chapter 1, Section 1.4. In addition, the performance can be further enhanced by adding an extra layer to the underlying neural network, emphasizing the significance of a well-designed architecture for the *Extended Network model*.

Here, we refer to the term "*Extended Network*" as an ensemble-like architecture, adaptable to accommodate diverse models and future methodologies as ensemble components. Potential extensions include various forms of *n-gram* models (such as cached or nested *n-gram* models) and decision trees. Promising directions for future research involve expanding ensemble sizes, optimizing model hierarchies, and fine-tuning underlying neural network architectures.

Nevertheless, while the component comparison reveals the effect of each component on the *Extended Network model*, it remains unclear how these components perform on specific prediction cases, such as keywords, parameters, or function names. Each of these cases holds varying importance in practical code completion for developers. This is attributed to the evaluation metric utilized in the experiments, which aggregates results across all types of code tokens. The following chapter introduces our methodology proposed to overcome this limitation.

Code Token Type Taxonomy

Chapter 5

A Methodology for Refined Evaluation of ML-based Code Completion Approaches

Code completion has become an indispensable feature of modern Integrated Development Environments (IDEs). In recent years, many approaches have been proposed to tackle this task. However, it is hard to compare between the models without explicitly re-evaluating them due to the differences of used benchmarks (e.g. datasets and evaluation metrics).

Additionally, almost all of these works report the accuracy of the code completion models as aggregated metrics averaged over all types of code tokens. Such evaluations make it difficult to assess the potential improvements for particularly relevant token types (i.e. method or variable names), and blur the differences between the performance of the methods.

This chapter introduces a methodology called *Code Token Type Taxonomy (CT3)* to address the issue of using aggregated metrics. Namely, we identified multiple dimensions relevant for code prediction (e.g. syntax type, context, length), partitioned the tokens into meaningful types along each dimension, and computed individual accuracies by type. We illustrated the utility of this methodology by comparing the code completion accuracy of a Transformer-based model in two variants: with closed, and with open vocabulary.

Our findings show that the refined evaluation provides a more detailed view on disparities and identifies areas requiring additional investigation. Besides, our review of the state-of-the-art of Machine Learning (ML)-based code completion models indicates the necessity for standardized benchmarks in this domain. Furthermore, we observed that the open vocabulary model exhibits notably higher accuracy for pertinent code token types, such as variable usage and literals.

Section 5.1 explains our motivation with illustrative examples, while Section 5.2 presents background information and related work. Our approach is detailed in Section 5.3, and the experimental evaluation is discussed in Section 5.4. In Section 5.5, we outline the challenges of developing *CT3* schema for Python, threats to validity, and our response to the third core research question (CRQ3). Additional statistics and results of tuning experiments are delineated in Section 5.6. Finally, we conclude the chapter in Section 5.7.

This chapter is based on our peer-reviewed publication (Le et al., 2023).

5.1 Introduction

Code completion is a widely used feature of modern IDEs, where the most likely next token is offered based on the code already present up to the cursor position (Kim et al., 2021). This

feature not only helps developers to save typing effort, but also assists them in learning new libraries, as it offers information about available functions or attributes.

ML approaches for code completion are leading the field, and in particular the Transformer models excel here by outperforming the Recurrent Neural Networks (RNNs). Multiple state-of-the-art solutions are using Transformers with variations of the code representation and/or the attention mechanisms (Kim et al., 2021; Liu et al., 2020; Svyatkovskiy et al., 2020).

Traditional evaluation metrics. A majority of the proposed code prediction approaches use aggregated metrics (i.e. averaged over all types of code tokens) to evaluate their accuracy (Li et al., 2018; Wang et al., 2021b; Chirkova et al., 2021). This eliminates valuable information about the improvements for relevant code token types.

Consequently, approach comparison or weakness identification in a method becomes more challenging when using aggregated metrics. A typical case of this elimination of information is the prediction results for variables and function/method calls contributing to the overall score together with less demanding but predictable tokens like keywords and standard libraries.

Example of refined accuracy versus aggregated accuracy. Figure 5.2 illustrates an example of detailed evaluation gained by using refined accuracy, in comparison to the traditional method of using aggregated accuracy (Figure 5.1). We employed a code snippet from the Python150k dataset[1] and omitted long strings for simplicity. The prediction results are obtained from the experiments in Chapter 4.

```
class TestServer(object):
    def __init__(self, application):
        self.application = application
        self.server = self.make_server(application)

    def get(self, *args, **kwargs):
        return self.request('get', *args, **kwargs)
```

(a)

Correct predictions	11/20
Incorrect predictions	9/20

(b)

Figure 5.1: An example for evaluating a code snippet using aggregated accuracy.

The incorrect predictions are highlighted in these figures by double-dashed-underlines. Various colors are also deployed as an additional discrimination. Statistics of correct and incorrect predictions are also presented in Figures 5.1(b) and 5.2(d). Since we only use a small code snippet as an example to demonstrate the advantage of the refined accuracy, the following interpretations are conjectures. The valid evaluation for the utilized completion model should be conducted on a proper dataset.

In general, while using aggregated accuracy, there are no hints for developers about where to improve the accuracy or to inspect the drawbacks of the used method. Meanwhile, using refined accuracy gives a more detailed view at the completion results for multiple dimensions (i.e. aspects of code token properties). Concrete examples are as follows:

Refined accuracy – token purpose. Firstly, considering the code tokens based on their purpose, we identified five token types for the code snippet as displayed in Figure 5.2(a): (i)

[1]SRILAB, https://www.sri.inf.ethz.ch/py150, (Accessed: 14 March 2024).

```
class TestServer(object):
    def ♦__init__(♥self, ♥application):
        self.application = application
        self.server = self.make_server(application)

    def ♦get(♥self, ♥*args, ♥**kwargs):
        return self.request('get', *args, **kwargs)
```

(a)

```
class TestServer(object):
    def __init__(self, application):
        self.application = application
        self.server = self.make_server(application)

    def get(self, *args, **kwargs):
        return self.request('get', *args, **kwargs)
```

(c)

```
class TestServer(object):
    def __init__(self, application):
        self.application = application
        self.server = self.make_server(application)

    def get(self, *args, **kwargs):
        return self.request('get', *args, **kwargs)
```

(b)

Token type	Correct pred.	Incorrect pred.
attribute	1/2	1/2
♥argument definition	2/5	3/5
variable usage	7/9	2/9
♦method definition	1/2	1/2
method call	**0/2**	**2/2**
short/medium token (*len≤10*)	9/15	6/15
long token (*len>10*)	2/5	3/5
high frequent token (`self`)	5/6	1/6

(d)

Figure 5.2: An example for evaluating a code snippet using refined accuracy.

attribute tokens in **bold**, (ii) variable usages in *italics*, (iii) method calls in ***bold and italics***, (iv) argument definitions beginning with a ♥, and similarly (v) method definitions with a ♦.

The statistics in Figure 5.2(d) show that nine out of 20 predicted tokens are variable usages and seven of those are suggested correctly. This can be considered as an advantage of the used completion model, since predicting usages of identifiers in general is significant for developers, according to Hellendoorn et al. (2019). However, the prediction of the other four token types still needs to be improved, especially for method call with no true completions.

Refined accuracy – token length. The next dimension analyzed in the example is token length. We considered code tokens with less than 11 characters as short or medium tokens and format them in ***bold and italics*** in Figure 5.2(b). The rest of the predicted code tokens are long tokens and marked as *italics*. The analysis results in Figure 5.2(d) reveal that more than 80% of the correct predictions belong to short/medium tokens, which slightly help reducing the typing effort of developers in this example.

Refined accuracy – token frequency. Eventually, we inspected the frequency of code tokens based on their number of repetitions in the example. The highly frequent tokens in Figure 5.2(c) (i.e. `self`) are highlighted in **bold**. Figure 5.2(d) shows that nearly half of the total correct predictions are high frequent tokens (5 out of 11 predictions), which is one of the easy cases of code completion. Other token frequency values are excluded for simplicity.

In conclusion, the above analysis brings valuable information, which cannot be obtained while using aggregated accuracy. This analysis leads to the idea of focusing on method calls and long-frequent tokens for improving the completion model, since predictions of other code token types already have acceptable accuracies. This example highlights how aggregated metrics hinder evaluation and development by omitting crucial information.

Refined evaluation. To our knowledge, only few previous works consider token categories and evaluate the metrics (e.g. mean reciprocal rank, error rate) per category, such as Bielik et al. (2016) and Kim et al. (2021). However, the token subdivision (e.g. attribute access,

numeric constant, string, variable/module name, and expression) in these works is rather crude, since they only focus on some cases of the token purposes and disregard other dimensions (e.g. length and frequency). Besides, it is difficult to apply their taxonomy to other works without re-implementation (further details in Section 5.2.2).

Contributions of this work. We aimed to provide more refined types of code tokens which can be used for evaluating existing and future code completion approaches with minimum effort. Our contributions in this work are as follows:

- *State-of-the-art of ML-based code completion models.* We reviewed these approaches (mostly from 2018 to 2021), not only to substantiate the usage of aggregated metrics but also to give an overview of the current research progress in this field. Despite of using the same modeling methods (e.g. Transformers) for the code completion task, it is still hard to compare between models without explicitly re-evaluating them. This is due to the differences on input representations (e.g. sequences of tokens or Abstract Syntax Trees), the employed datasets, evaluation metrics (e.g. accuracy, MRR), and the scope of prediction (e.g. next n code token or block of code).

- *Code Token Type Taxonomy.* To obtain a refined evaluation of code completion accuracy, we introduced a methodology called *Code Token Type Taxonomy (CT3)* by proposing multiple dimensions for identifying code token types. For each dimension, we proposed the types by analyzing the Abstract Syntax Tree (AST) and the relationships between tokens in the AST. *CT3* can be used for a comprehensive comparison between approaches, to gain a detailed view of the impact of each component in a prediction model, and to identify model challenges. The original version of the CT3 schema was developed by Rashidi (2021) in his Bachelor's thesis conducted within our research group.

- *Empirical study.* We illustrated the utility of this methodology by conducting an empirical study on the Python150k dataset of a Transformer-based code completion approach. We compared the impact of using closed vocabulary versus open vocabulary (Karampatsis et al., 2020), and found significantly better accuracy of the latter for relevant token types.

- *Source code and data.* To facilitate reproducibility and reuse of our methodology, we published the Python150k dataset with pre-computed token types[2] according to *CT3*. The scripts of our experiments and *CT3* source code[3] are also available.

Besides the main concern of enhancing code completion evaluation, which directly addresses the *programming barrier* </>, our published source code and data facilitate assessment on alternative code completion approaches, thereby alleviating the *reuse problem*. Furthermore, the implementation of *CT3* functions in parallel partially resolves the *scalability problem*. These issues are defined in Chapter 1, Section 1.1. Ultimately, the outcomes of this chapter respond to the third core research question (CRQ3), specified in Section 1.4 of Chapter 1.

[2] Code token type data, https://doi.org/10.5281/zenodo.5733013.
[3] GitLab repository, https://gitlab.com/pvs-hd/published-code/code-token-type-taxonomy (Accessed: 14 March 2024).

5.2 Background and Related Work

In this section, we present the state-of-the-art of code completion, followed by an overview of the usage of aggregated and refined metrics for evaluating code completion models in previous research. We end the section with a brief introduction to the Out-of-Vocabulary (OOV) issue and its current possible solutions.

Table 5.1 summarizes the notable works in code completion mostly from 2018 to 2021. We divided the prediction level (the fourth column of the table) into: (i) *token*, the possible next token, (ii) *tokens*, the next n tokens up to the end of the code line, (iii) *code line*, the entire code line, (iv) *construct*, specific code constructs, e.g. an `if` condition, and (v) *block*, code blocks, e.g. a `for` loop or an entire function.

Besides, the programming languages used in datasets are documented in the "*Dataset*" column (except for the widespread datasets Python150k and JavaScript150k[4]). We refer to the original papers for more details. The column "*Eval. ter. & non-ter.*" shows how authors handled the results on terminal and non-terminal nodes in ASTs, which is only applicable when the representation of input data is AST-related forms.

The table is sorted by year and grouped into three parts: recent methods (from RNN to Codex), popular tools (from Kite to GitHub Copilot), and two potential solutions for the OOV issue (the last two rows). Due to space constraints, citations are omitted from the table and presented in the following subsections. Additional details, such as evaluation results and handling of node types and values in ASTs, are included in an online version of the table[5].

Table 5.1: State-of-the-art of code completion models, primarily from 2018 to 2021.

Tool name	Modeling method	Year	Pred. level	Dataset	Eval. metrics	Input form	Eval. ter. & non-ter.‡
RNNs	RNNs, n-gram	2014	code line	Java	MRR, accuracy	seqs. tokens	N/A
n-gram	n-gram	2016	token	C, Java	perplexity, cross-entropy	seqs. tokens	N/A
PHOG	PCFG	2016	token	JavaScript 150k	error rate, log-probability, ***categories***	AST nodes	separated results
Pointer Mixture Network*	RNNs, Pointer Network	2017	token	Python150k, JavaScript 150k	accuracy	AST nodes	separated results
CodeGRU*	GRU	2020	token, tokens	Java	MRR, accuracy	seqs. tokens†	N/A

continued ...

[4]SRILAB, https://www.sri.inf.ethz.ch/js150.
[5]SOA Code Completion, https://doi.org/10.5281/zenodo.5739285. (Accessed: 15 March 2024).

. . . continued

Tool name	**Modeling method**	**Year**	**Pred. level**	**Dataset**	**Eval. metrics**	**Input form**	**Eval. ter. & non-ter.**‡
Structural Language Modeling*	LSTM, copy mechanism	2020	construct	Java, C#	accuracy, tree@k	AST paths	integrated results
Extended Network model	PHOG, Pointer Mixture Network	2020	token	Python150k	accuracy	AST nodes	terminal only
IntelliCode Compose*	Transformers	2020	tokens	Python, C#, JavaScript, TypeScript	perplexity, ROUGE-L, Levenshtein	seqs. tokens	N/A
Feeding Trees to Transformers*	Transformers	2021	token	Python150k, Facebook internal repositories	MRR, ***categories***	AST nodes, seqs. tokens	integrated & separated results, local Beam search
CCAG	GNN	2021	token	Python50k, JavaScript 150k	accuracy	AST graph	separated results
Transformers for Source Code*	Transformers	2021	token	Python 150k,** JavaScript 150k	MRR	AST nodes	separated results
BERT for Code Completion	RoBERTa	2021	tokens, construct, block	Java, Android	BLEU, Levenshtein, perfect prediction, semantic equiv.	seqs. tokens	N/A
Transformers for Code Completion	Transformers	2021	tokens, construct, block	Java, Android	BLEU, Levenshtein, perfect prediction	seqs. tokens	N/A
Codex*	GPT-3	2021	block	Python, APPS	pass@k	doc-strings	N/A
Kite	GPT-2	2017	tokens				
TabNine	GPT-2	2019	tokens				
GitHub Copilot	Codex	2021	block				
Byte-pair Encoding for OOV*	BPE	2020	token	Java, C, Python	MRR, cross entropy	seqs. tokens	N/A

continued . . .

. . . continued

Tool name	Modeling method	Year	Pred. level	Dataset	Eval. metrics	Input form	Eval. ter. & non-ter.‡
Anonymization for OOV*	Transformers	2020	token	Python 150k,** JavaScript 150k**	MRR	AST nodes	integrated results

* Source code is available.
** The redistributable version of the Python150k or JavaScript150k dataset.
† Each identifier is replaced by its datatype.
‡ Evaluation results on terminal and non-terminal nodes.

5.2.1 Machine Learning for Code Completion

ML adoption. State-of-the-art approaches for code completions or general code predictions utilize ML-based techniques (Le et al., 2020). Methods include RNNs (Raychev et al., 2014), *n-gram* language models (Hindle et al., 2016), context-aware Gated Recurrent Unit (GRU) by Hussain et al. (2020) or context-aware Convolutional Neural Network (CNN) of Hussain et al. (2021), Probabilistic Higher Order Grammar (PHOG) from Bielik et al. (2016), Pointer Mixture Network (Li et al., 2018), and Structural Language Modeling (Alon et al., 2020b), or hybrid approaches as our *Extended Network* (Chapter 4).

The emergence of Transformers. Recent works such as IntelliCode Compose (Svyatkovskiy et al., 2020), Feeding Trees to Transformers (Kim et al., 2021; Liu et al., 2020) use Transformer models (Vaswani et al., 2017), which outperform RNNs. One of the reasons is that Transformers better capture the long-range dependencies.

There are several empirical studies on the capabilities of ML-based code completion models, which use Transformers and its variants for experiments, including Transformers for Source Code (Chirkova et al., 2021), BERT for Code Completion (Ciniselli et al., 2021b) and Transformers for Code Completion (Ciniselli et al., 2021a). Further details on Transformers are discussed in Chapter 2, Section 2.3.

Generative Pre-trained Transformer 2 and 3 (GPT-2 and GPT-3) are also well-known methods in this domain. Kite[6] and Tabnine[7], which are based on GPT-2, are noteworthy tools supporting code completion for multiple programming languages. Chen et al. (2021) proposed a model named Codex based on GPT-3 for generating Python functions from docstrings. They also introduced a tool named GitHub Copilot[8].

There are also other works that focus on other modeling methods rather than Transformers. Wang et al. (2021b) proposed a model named CCAG based on Graph Neural Network (GNN) to fully capture the sequential and repetitive patterns of code, together with the structural

[6] Kite stopped supporting since November 2022, https://www.kite.com/.
[7] AI coding assistant, https://www.tabnine.com/.
[8] AI developer tool, https://github.com/features/copilot/.
(Accessed: 15 March 2024).

information on ASTs. The evaluation results show that CCAG outperforms state-of-the-art approaches, including Transformers.

A need for standardized benchmarks. In general, ML-based models for code completion have recently attracted a lot of attention. However, it is still hard to fairly compare between the models, since many approaches use proprietary evaluation benchmarks. Two models are comparable if they at least use the same evaluation metrics and datasets. For instance, considering the models using the *accuracy* metric and the *Python150k* dataset, we narrow down the list of models in Table 5.1 to Pointer Mixture Network, *Extended Network model*, and CCAG.

These models have the same prediction level, a similar form of input data and the same way of separating results for terminal and non-terminal nodes. Still, *Extended Network model* and CCAG can not be compared to each other as they use different experimental setups and various sizes of test datasets. The two remaining comparisons are presented explicitly in papers of the authors by re-evaluating all models. Hence, there should be a set of standardized benchmarks for code completion models to make the models comparable with a reasonable effort.

5.2.2 Aggregated and Refined Metrics for Evaluation

Table 5.1 discloses that almost all prediction results use aggregated metrics to evaluate the accuracy (i.e. averaging over all code token types). Exceptions are Kim et al. (2021) and Bielik et al. (2016) providing a rough analysis, which is indicated by value ***categories*** in column "*Eval. Metrics*" of Table 5.1. However, their evaluations only focus on some general values of code token purposes.

Criteria for a Refined Evaluation

Hellendoorn et al. (2019) revealed in their study that there are large differences which code completions are relevant when considering the point of view of developers versus the distributions in synthetic datasets of completions. Their results and insights inspired us to determine criteria for a refined evaluation in code completion. Table 5.2 presents the criteria together with their corresponding motivations.

For example, instead of punctuation-like tokens, identifier token type represents code completions most relevant for developers (Hellendoorn et al., 2019). This finding motivated us to analyze syntax type information of code tokens in prediction results. Moreover, based on our experimental observation (presented in Section 5.4.3), it is hard to predict the identifier names in the declaration due to the arbitrariness of identifiers.

However, only a small portion of code tokens in the dataset is categorized as this syntax type. In other words, distinguishing between definition and usage of identifiers is essential. Therefore, values of the dimension syntax type should cover not only token purposes but also support dividing between definition and usage of identifiers.

Additionally, the importance of local context, typing effort, origin, and rare completions (i.e. long and infrequent tokens) is also highlighted by Hellendoorn et al. (2019). Their insights motivated the criteria of our context, length, origin, and frequency dimensions, respectively,

Table 5.2: Criteria for a refined evaluation.

Dimension	Criteria for values of the dimension	Motivation
Syntax Type	Covering relevant code token types with a sufficient level of details	Identifier token type is the most demanded completion for developers
	Supporting the division between definition and usage of identifiers	Distinguishing between definition and usage of identifiers is essential*
Context	Maximizing the detail regarding the surrounding structures of a code token.	Local context plays a large role in code completion accuracy
Origin	Indicating the location where the token is defined, i.e. from the same file, from external/standard/built-in libraries	Method invocations within the same project is the most prominent sub-category in datasets
Length	Reflecting the length of a code token in comparison with other tokens in a dataset	The longer the code token, the higher the likelihood of completion request.
Frequency	Expressing the frequency of occurrence of a code token in an AST	Rare and difficult completions (i.e. long and not so frequent code tokens) are vital under the view of developers

* This result is derived from our experimental observation (details in Section 5.4.3), while the others are findings of Hellendoorn et al. (2019).

as expressed in Table 5.2. Analyzing completion results with aggregated metrics apparently dissatisfies these criteria.

Refined Evaluation in Previous Studies

To clarify that none of the prior works provide evaluations with a sufficient level of details, Table 5.3 presents a qualitative comparison between the methods of Kim et al. (2021), Bielik et al. (2016), and our methodology based on the defined criteria. We considered one more criterion on the implementation of these approaches (last column of Table 5.3). Namely, the refined evaluation should be implemented in a way that token types data can be combined with completion logs obtained from prediction models, with a reasonable effort.

Table 5.3: *CT3* and related works in supporting refined evaluations.

Evaluation methodology	Dimensions					Versatility across completion models*
	Syntax Type	Context	Origin	Length	Frequency	
Bielik et al. (2016)	◐	○	○	○	○	○
Kim et al. (2021)	◐	◐	○	○	○	●
CT3 (ours)	●	●	●	●	●	●

○ The criterion is not supported.
◐ The methodology considers some values in the dimension but does not meet all criteria of the dimension.
● The criterion is accomplished fundamentally.
* With a reasonable effort.

Bielik et al. only considered some specific values in the dimension syntax type (e.g. identifier,

property, and number). They lacked a refined categorization for identifiers, such as function, variable, and class, as well as the separation between definition and usage of identifiers. Besides, the authors implemented an individual predictor for each token type, which makes it hard to apply their evaluation approach to other prediction models.

Meanwhile, **Kim et al.** provided subcategories for identifiers (i.e. attribute access, variable/module name, and function parameter name). However, the subdivision is rather crude and still leaves out several important types, e.g. function/method definition, class definition, and imported library. The published source code from their paper[9] shows that the authors might group attribute, class usage, and method usage into the attribute category. Similarly, variable/module name might contain variable, class usage, and function usage.

Grouping different code token purposes into one group might omit crucial information because of the diverse effect of various code token types on the efficiency of completion models (further details in Section 5.4.3). In addition, the authors also examined some values in the dimension context but still ignored some essential contexts, e.g. in-class-definition or in-function-definition. Other dimensions (i.e. origin, length, and frequency) are not supported in their methodology.

Ultimately, the authors provided scripts to identify token IDs for each of their token types, which can be applied to other completion models. However, their short Python scripts comprise 119 LOC and consider only two categories, i.e. dimension syntax type with four values and dimension context with six values. In addition, these values are derived directly from value or type of a target node in an AST, regardless of whether the node is a terminal or non-terminal.

CT3. We proposed a methodology to effectively evaluate code completion models and assure all the mentioned criteria. Our prototypical implementation for Python covers 18 values for syntax type dimension, 16 values for context dimension, four values for origin dimension, and dimensions length and frequency both contain three distinct values. The syntax type dimension comprises significant token types as well as a subdivision of definition and usage of identifiers.

Additionally, we constructed our implementation in an extendable way, in terms of number of dimensions, number of values of each dimension, and independence of prediction models. Furthermore, values between dimensions can be combined to create an even more complicated code token type (e.g. determining usages of imported libraries requires both dimensions syntax type and origin, discussed in Section 5.5).

Furthermore, we provided a sophisticated analysis for each value of dimensions, by considering not only the target node but also its neighbors and ancestors along the path to the root node in ASTs. We separated terminal and non-terminal nodes, since we only evaluated code tokens that developers will receive from a completion model (i.e. values of terminal nodes).

Finally, we published pre-computed data of token types, which can be utilized with completion logs created from various prediction models considering ASTs as input data. The refined evaluation results obtained from our proposed methodology might facilitate comparison and improvement of prediction models for relevant cases. A detailed explanation with examples for the proposed approach is presented in Section 5.3.

[9] We only compare the implementation of model *trav_trans* and our implementation for *CT3* since both methods consider ASTs as input data.

5.2.3 Out-of-Vocabulary Issue

An important aspect of the prediction approaches is code representation. While some works use as input a sequence of AST nodes linearized by a tree traversal (Bielik et al., 2016; Li et al., 2018; Schumacher et al., 2020), more recent approaches attempt to capture the high-level structural representation (Alon et al., 2020b; Kim et al., 2021). Besides, Chirkova et al. (2021) indicated that only syntactic information is needed to make meaningful predictions.

Another crucial factor of code representation is capturing the code identifiers. Even with coding conventions, identifiers defined by software developers can at times be overly diverse and complex. As a result, prediction models might be trained with an exceedingly large and sparse vocabulary. Furthermore, if the size of the vocabulary is limited, rare words in the training set have a low possibility to contribute to the vocabulary and therefore are hard to be predicted. This is also known as the Out-of-Vocabulary (OOV) issue.

With the original method, the vocabulary of a prediction model is constructed on a fixed set of tokens, commonly based on the frequency of tokens in the dataset, which creates the OOV issue. Karampatsis et al. (2020) suggested encoding tokens with an open vocabulary via Byte-Pair Encoding (BPE), which is considered as a prominent solution for this problem.

While only few works use the open vocabulary model, e.g. Svyatkovskiy et al. (2020), the results of our work show that open vocabulary can significantly improve the accuracy of relevant token types. Besides, Kim et al. (2021) also mentioned open vocabulary as one of directions to enhance their research works in future.

Alternatively, Chirkova et al. (2020) introduced another approach to deal with the OOV issue by anonymizing all the OOV tokens with special placeholders. However, since these approaches use different types of input representation and datasets, the comparison of their efficiency needs further investigation and is out of scope of our work.

5.3 Code Token Type Taxonomy

This section first presents our proposed methodology – *Code Token Type Taxonomy (CT3)* – for a refined evaluation, followed by our approach of using open vocabulary to alleviate the OOV issue and to demonstrate the utility of *CT3*.

5.3.1 General Workflow

Figure 5.3 illustrates the implementation and usage of *CT3*.

Determination of CT3 schema. Initially, given a programming language, we identified relevant code token properties for effective code completions. This gives rise to multiple dimensions (i.e. criteria for partitioning) and the token types within each dimension (i.e. a complete subdivision of all tokens into types). Table 5.4 shows such a schema for Python. Subsequently, a static code analyzer is implemented.

Construction of CT3-enhanced dataset. Given a dataset (e.g. Python150k), the *CT3* code analyzer assigns to each code token a type for each dimension. Table 5.5 displays an example

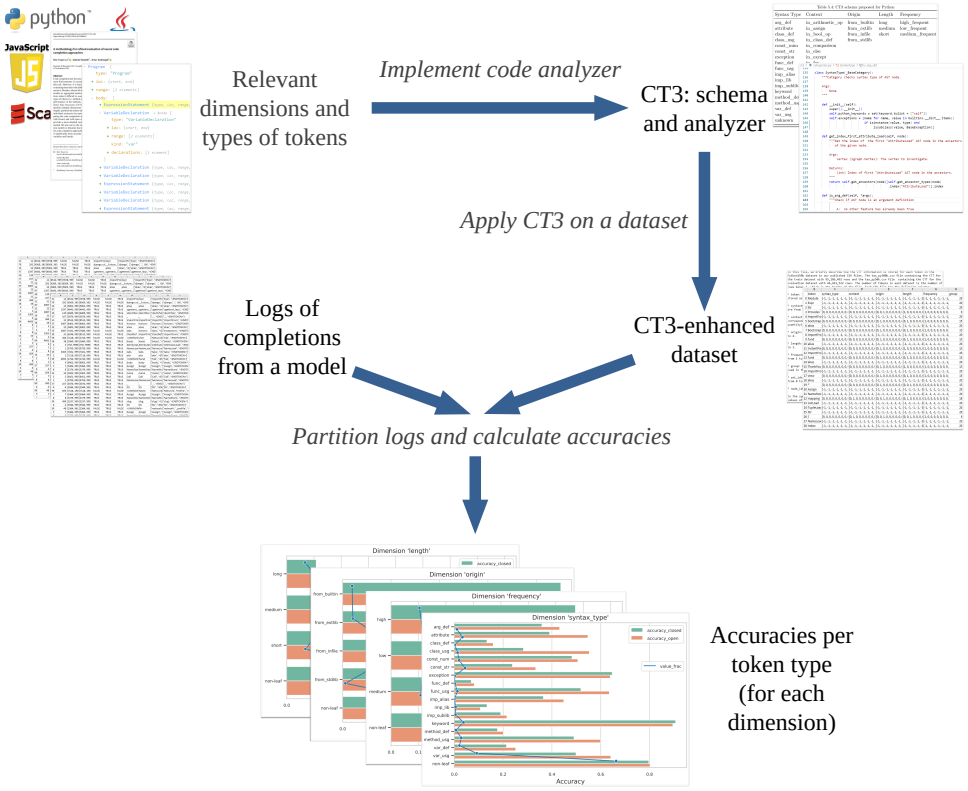


Figure 5.3: Implementation and usage of *Code Token Type Taxonomy (CT3)*.

Table 5.4: *CT3* schema proposed for Python.

Syntax Type	Context	Origin	Length	Frequency
arg_def	in_arithmetic_op	from_builtin	long	high_frequent
attribute	in_assign	from_extlib	medium	low_frequent
class_def	in_bool_op	from_infile	short	medium_frequent
class_usg	in_class_def	from_stdlib		
const_num	in_comparison			
const_str	in_else			
exception	in_except			
func_def	in_for			
func_usg	in_func_def			
imp_alias	in_if			
imp_lib	in_parameter			
imp_sublib	in_raise			
keyword	in_return			
method_def	in_try			
method_usg	in_while			
var_def	in_with			
var_usg				
unknown				

of *CT3* information for code tokens in the Python150k dataset. Values of the dimensions *Syntax Type*, *Origin*, *Length*, and *Frequency* are illustrated by their indices in the lists of possible values (list index starts with 1, Table 5.4).

Notably, a special value −1 is used to indicate that the code token is a non-terminal node or

Table 5.5: An example of *CT3* data for code tokens in the Python150k dataset.

Token	Syntax Type	Context	Origin	Length	Frequency	ASTIdx	NodeIdx
`ImportFrom`	−1	−1	−1	−1	−1	0	1
`django.utils.translation`	11	`[0]`	2	1	1	0	2
`alias`	−1	−1	−1	−1	−1	0	3
`ugettext_lazy`	12	`[0]`	2	1	1	0	4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
`ClassDef`	−1	−1	−1	−1	−1	0	11
`NetworkProfileTab`	3	`[(4,1)]`	3	1	1	0	12
`bases`	−1	−1	−1	−1	−1	0	13
`AttributeLoad`	−1	−1	−1	−1	−1	0	14
`NameLoad`	−1	−1	−1	−1	−1	0	15
`tabs`	4	`[(4,1),(11,1)]`	2	2	1	0	16
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

its token type is not under our consideration. The values of the *Context* dimension are displayed as a list of tuples `(context_index, quantity)`, which is exemplified later on. The value −1 is also used to denote the context value of non-terminal nodes. However, if a terminal node does not belong to any context we are focusing on (outlined in Table 5.4), then its context value will be `[0]`. Besides, we also recorded the indices of nodes in an AST and the AST indices in a dataset (for easily tracking back to original code files).

For instance, the code token `ClassDef` is a non-terminal node so its token types are −1 for all dimensions. Meanwhile, the token `tabs` is a terminal node and a base class in a class definition (i.e. `class NetworkProfileTab(tabs)`). Its token types are: 4/*class_usg* for *Syntax Type*, 2/*from_extlib* for *Origin*, 2/*medium* for *Length*, and 1/*high_frequent* for *Frequency*. Ultimately, the value `[(4,1),(11,1)]` of the *Context* dimension illustrates that `tabs` is in the context of *in_class_def* once and *in_parameter* once. Further details of the *CT3*-data and the storage formats are presented in our published dataset[10].

Combination of code completion log and CT3-data. Following the workflow depicted in Figure 5.3, the *CT3*-enhanced data is merged with logs of code completions generated by a prediction model (middle section of Figure 5.3). The resulting log file includes *CT3*-data for each code token along with prediction information. An illustration of such a log file is provided in Table 5.6. The outcomes are derived from Transformer-based models employing closed and open vocabularies.

For example, the token `tabs` is predicted correctly in both closed and open vocabulary cases while only the first part of the `django.utils.translation` token can be predicted by the open vocabulary variant. Two special symbols `<UNKNOWN>` and `<ENDTOKEN>` are used to indicate tokens that cannot be encoded by the closed/open vocabulary models and to mark the end of sequences of subtokens, respectively. More details of encoding code tokens using closed and open vocabulary with Transformer-based models are discussed in Section 5.4.2.

The example in Table 5.6 also shows that *CT3*-data can be combined with any completion logs, as long as they can specify the order of evaluated code tokens in the dataset[11]. This makes our methodology more flexible than other related works, as mentioned in Section 5.2.2.

[10] Code token type data, `https://doi.org/10.5281/zenodo.5733013`, (Accessed: 15 March 2024).

[11] In our experiments, we assigned to each code token a unique `ID` and used it as a key to map the *CT3*-data to the completion log.

Table 5.6: An example of a combination log of *CT3*-data and prediction results for code tokens in the Python50k dataset.

Closed vocabulary case		Open vocabulary case		Syntax Type	Context	Origin	Len.	Freq.	AST Idx	Node Idx
True symbol	Predicted symbol	True subtokens	Predicted subtokens							
ImportFrom	Expr	[‘ImportFrom’, ‘<ENDTOKEN>’]	[‘Expr’, ‘<ENDTOKEN>’]	−1	−1	−1	−1	−1	0	1
django.utils.translation	__future__	[‘django’, ‘.’, ‘utils’, ‘.’, ‘translation’, ‘<ENDTOKEN>’]	[‘django’, ‘.’, ‘db’, ‘<ENDTOKEN>’]	11	[0]	2	1	1	0	2
alias	alias	[‘alias’, ‘<ENDTOKEN>’]	[‘alias’, ‘<ENDTOKEN>’]	−1	−1	−1	−1	−1	0	3
ugettext_lazy	ugettext_lazy	[‘ugettext_lazy’, ‘<ENDTOKEN>’]	[‘ugettext_lazy’, ‘<ENDTOKEN>’]	12	[0]	2	1	1	0	4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
ClassDef	ImportFrom	[‘ClassDef’, ‘<ENDTOKEN>’]	[‘ImportFrom’, ‘<ENDTOKEN>’]	−1	−1	−1	−1	−1	0	11
<UNKNOWN>	User	[‘Network’, ‘Profile’, ‘Tab’, ‘<ENDTOKEN>’]	[‘Tab’, ‘<ENDTOKEN>’]	3	[(4,1)]	3	1	1	0	12
bases	bases	[‘bases’, ‘<ENDTOKEN>’]	[‘bases’, ‘<ENDTOKEN>’]	−1	−1	−1	−1	−1	0	13
AttributeLoad	AttributeLoad	[‘AttributeLoad’, ‘<ENDTOKEN>’]	[‘AttributeLoad’, ‘<ENDTOKEN>’]	−1	−1	−1	−1	−1	0	14
NameLoad	NameLoad	[‘NameLoad’, ‘<ENDTOKEN>’]	[‘NameLoad’, ‘<ENDTOKEN>’]	−1	−1	−1	−1	−1	0	15
tabs	tabs	[‘tabs’, ‘<ENDTOKEN>’]	[‘tabs’, ‘<ENDTOKEN>’]	4	[(4,1), (11,1)]	2	2	1	0	16
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

`Predicted symbol` is obtained with closed vocabulary.
`Predicted subtokens` are generated with open vocabulary.

Calculation of accuracies per token type. In the final step of the workflow, the log is partitioned according to the types per dimension. Next, the desired evaluation metric (e.g. accuracy) is computed for each type of dimensions. Examples for this step are discussed in Section 5.4.3. Our prototypical implementation for Python analyses the AST and the relationships between its nodes. It is worth to recall that we consider only terminal (leaf) nodes in the AST.

5.3.2 CT3 Schema for Python

This subsection explores the *CT3* schema tailored for Python, as exhibited in Table 5.4.

Syntax Type Dimension

The syntax type dimension refers to the syntactic category of a token in the source code. Values of this dimension (i.e. first column in Table 5.4) can be generalized for various programming languages and offer information regarding the code token's purpose. The majority of these values describe identifiers since predicting them is the most relevant function of code completion in practice (Hellendoorn et al., 2019).

Syntax type values are derived based on the syntactic information on the source code and the patterns in ASTs. The difficulty of identifying these token types varies greatly. Simpler types

mostly depend on conditions of identifying AST node types. Complex types are aggregations of conditions which identify feature-specific AST patterns. We defined 18 *labels* for this dimension, outlined as follows:

The *arg_def* token type refers to code tokens which are arguments in definitions of functions or methods. These code tokens can be regular arguments, keyword arguments, or `*args` and `**kwargs` in Python. Code tokens which are attributes of packages or classes have *attribute* as their syntax type.

Definitions and usages of classes are represented by *class_def* and *class_usg* token types, respectively; analogously for functions, methods, and variables (i.e. *func_def* and *func_usg*, *method_def* and *method_usg*, *var_def* and *var_usg*, respectively). While the token types *func_usg* and *method_usg* indicate function and method calls, the type *class_usg* expresses usages of defined classes, e.g. a class instantiation or a class being used as an inheritance parameter. The *var_usg* comprises usages of defined variables, declared arguments, imported libraries, and function keywords in function/method calls.

In the code snippet in Listing 5.1, the tokens `get_info` and `std_id` in line 1 are identified as *func_def* and *arg_def* types. While the variables `student` (line 2), `s_name` (line 3), and `s_class` (line 4) are classified as the *var_def* token type, the `std_id` (line 2) and `student` (line 3) variables are grouped into the *var_usg* type. The tokens `Student` and `StudentClass` are labeled as the *class_usg* type. Ultimately, the *attribute* token type is used to denote the remaining tokens, i.e. `profile` and `name` in line 3 as well as `info` and `name` in line 4.

```
1 def get_info(std_id):
2   student = Student(std_id)
3   s_name = student.profile.name
4   s_class = StudentClass.info.name
```

Listing 5.1: An example code snippet for extracting syntax type information.

The syntax type dimension also contains numeric and literal constants (*const_num* and *const_str* token types) and keywords defined for the target programming language (*keyword* type). Code tokens referring to exceptions defined in `try/catch` blocks are assigned to the *exception* token type, which covers all built-in exceptions.

Ultimately, token types *imp_lib*, *imp_sublib*, and *imp_alias* are used to identify imported libraries, imported sublibraries, and their aliases, respectively. The 18th value of the dimension (i.e. *unknown*) indicates tokens which can not be categorized in other values. Most of these tokens are empty lists of parameters. These lists are still presented in ASTs but don't affect the code completion accuracy. Table 5.7 summarizes the explanation and examples for each label of the syntax type dimension, with the target code token highlighted in every example.

Context Dimension

The contextual information describes the surrounding code structures (e.g. loop body, condition expression) in which the token is found. The context values (i.e. second column in Table 5.4)

Table 5.7: Explanation and examples for Syntax Type dimension of *CT3*.

Type	**Explanation**	**Example for each *type***
arg_def	A definition of an argument in a function or a method definition	`def a_func(arg1,arg2="value", *args,**kwargs)`
attribute	An attribute of a package or a class	`AClass.attribute1.attribute2`
class_def	A class definition	`class Aclass`

class_usg	A usage of a class	`class AnotherClass(BaseClassUsage)`
		`a_var = ClassUage()`
		`Lib.ClassUsage()`
		`a_var = ClassUsage.an_attribute`
		`results = ClassUsage.a_method()`

const_num	A numeric constant	`a_var = 10`
const_str	A literal constant	`a_var = "a string"`
exception	An exception value	`except AnException:`
func_def	A function definition	`def a_func(...):`
func_usg	A function call (only for functions defined within the same file)	`results = func_usg(...)`
imp_alias	An alias of an imported library	`import a_lib as an_alias`
imp_lib	An imported library	`import a_lib`
imp_sublib	An imported sublibrary	`from a_lib import a_sublib`
keyword	A keyword of the target programming language	`a_var = True`
method_def	A method definition	`def a_method(...):`
method_usg	A method call (including function calls from imported libraries)	`AClass.method_usg(...)`
var_def	A variable definition, including local variables (except the ones in `lamda` functions, `if`/`else` statements and loops) and global variables	`a_var = "a variable"` `global a_glob_var`

var_usg	A usage of a defined variable	`a_var += 1`
		`results = a_func(a_var)`
		`results = a_var[idx]`
		`results = a_var.func_call()`
		`std_name = student.name`
		`results = a_func(var1, kw_var=var2)`

unknown	The remaining cases	Empty list of parameters

aim to reflect the local context, which plays a large role in code completions (Hellendoorn et al., 2019). We proposed 16 *values* corresponding to different levels of code structure.

The values *in_arithmetic_op*, *in_assign*, *in_bool_op*, and *in_comparison* indicate code tokens found in arithmetic, assignment, boolean, and comparison operations, respectively. Code tokens located in class or function/method definitions are represented by the context values *in_class_def* or *in_func_def*. The parameters used in these definitions, as well as in function/method calls, are assigned to the *in_parameter* value.

Code tokens being used in `if` or `else` statements are labeled with the *in_if* or *in_else* values; analogously for *in_try*, *in_except*, and *in_raise*. The values *in_for* and *in_while* refer to the tokens occurring in loop structures (i.e. `for` and `while`). Finally, *in_return* expresses tokens in `return` statements, while *in_with* indicates code tokens within `with` statements.

Since the code structures represented by these context values can be nested in some cases, we recorded in how many contexts of a given value each code token is included. Listing 5.2 illustrates the token `var_example` (line 7) which is in the context of *in_class_def*, *in_func_def*, *in_if*, *in_for* (twice), *in_try*, *in_parameter*, and *in_arithmetic_op*. Comprehensive examples for each context value are provided in Table 5.8. For simplicity, we outlined common code token positions in these examples, as all tokens within the scope share identical contextual values.

```
1  class ClassDef():
2    def func_def(self, ...):
3      if ... :
4        for i in ...:
5          for j in ...:
6            try:
7              a_func(var_example + ...)
8            except ... :
9              ...
10     else:
11       ...
```

Listing 5.2: An example code snippet for extracting context information.

Origin Dimension

The origin labels (i.e. third column in Table 5.4) indicate the location where an identifier or a keyword is defined. A code token's origin can provide valuable information about its purpose and importance. While built-in code tokens (e.g. keywords or methods available without importing) are frequently used and have a rather general purpose, code tokens originating from within the same file serve a specific purpose which is closely related to the task the code solves.

The *from_builtin* label represents built-in code tokens, which do not require an explicit import such as keywords (e.g. `True`/`False`). Tokens categorized as *from_extlib* originate from external (non-standard) libraries or packages. Identifiers defined in the same file have *from_infile* as their origin label. Ultimately, the *from_stdlib* label refers to identifiers defined in standard libraries.

The above labels are obtained by analyzing the `import` commands in each AST. Built-in code tokens are those within a predefined Python keyword set. Tokens from within the file are determined by exclusion, as these tokens are neither built-in nor from the standard library or an external library.

Besides, code tokens appearing as attributes of a particular library are categorized according to the origin information on the library. For example, considering the code line 4 in Listing 5.1, if the token `StudentClass` is assigned to the *from_infile* label, then the `info` and `name` tokens also have *from_infile* as their origin label.

Length Dimension

Values of the length dimension denotes the number of characters in a code token. This dimension is motivated by the fact that long code tokens benefit more from code completions than short ones, since they save more typing effort (Hellendoorn et al., 2019). The length also correlates

Table 5.8: Explanation and examples for Context dimension of *CT3*.

Value	**The code token in ...**	**Example for a specific *token***
in_arithmetic_op	an arithmetic operation	`result = token + 10`
in_assign	an assign operation	`token = value`
in_bool_op	a boolean operation	`token != another_value`
in_class_def	a class definition	`def AClass:` `  token = "a string"`
in_comparison	a comparison operation	`token < 50`
in_else	an `else` statement	`else:` `  token = value`
in_except	an exception block	`except ValueError:` `  token = "another string"`

in_for	a `for` loop	`for token in an_array:  ...`
		`for i in range(5):`
		`  token = value`

in_func_def	a function definition	`def a_func(...):` `  token = value`

in_if	an `if` statement	`if token != 1:  ...`
		`if a_var > 1:`
		`  token = value`

in_parameter	a parameter list	in function/method definitions and calls, or in class definitions/instantiations
in_raise	a `raise` statement	`raise token`
in_return	a `return` statement	`return token`
in_try	a `try` statement	`try:` `  token = value`

in_while	a `while` loop	`while token > 0:  ...`
		`while a_var < 100:`
		`  token = value`

in_with	a `with` statement	`with open(file, "a") as token:`

with the importance of a code token. Short tokens usually hold temporary values (e.g. `i` as a loop counter), which are less significant.

Code tokens are categorized as *short* if containing up to 3 characters, *medium* for 4 to 10 characters, and *long* for lengths exceeding 10 characters. These thresholds are determined from statistical analysis of terminal lengths within the Python150k dataset (described in Section 5.6.1). Approximately 19.7% of terminal tokens in the dataset are 4 characters long, with this proportion decreasing notably for longer lengths. Moreover, among the top-15 most common terminal token lengths in the dataset, the majority fall within the 4 to 10 character range.

Frequency Dimension

Finally, frequency labels refers to a code token's frequency, relative to the frequency distribution of all code tokens within each individual AST. We used three values here: *low*, *medium*, and *high*, based on intervals explained in Figure 5.4. On one hand, long and frequent code tokens are

likely to be significant. On the other hand, while short code tokens can be frequent, in general they carry insignificant (e.g. temporary) values.

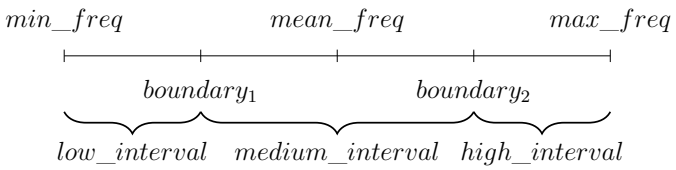


Figure 5.4: Frequency intervals for determining frequency labels of code tokens with *CT3*, adapted from Figure 4.3 of Rashidi (2021).

The three intervals specifying the frequency of occurrence as *low*, *medium*, and *high* are adjusted based on the *min*, *mean*, and *max* values of the frequency distribution of all tokens within the AST. Equations 5.1a to 5.1e show the calculation for these intervals. Except the *high_interval*, which is a closed interval, the *low_interval* and *medium_interval* are half-open intervals (i.e. the higher endpoints are not included, Equations 5.1c and 5.1d).

$$boundary_1 = \frac{mean\_freq - min\_freq}{2} \tag{5.1a}$$

$$boundary_2 = \frac{max\_freq - mean\_freq}{2} \tag{5.1b}$$

$$low\_interval = [min\_freq, boundary_1) \tag{5.1c}$$

$$medium\_interval = [boundary_1, boundary_2) \tag{5.1d}$$

$$high\_interval = [boundary_2, max\_freq]. \tag{5.1e}$$

5.3.3 Open Vocabulary for Transformers

Since Transformers recently gained a lot of attention with promising results on the code completion task (Svyatkovskiy et al., 2020; Kim et al., 2021; Chirkova et al., 2021; Ciniselli et al., 2021a), we chose Transformer-based models as representative models for the state-of-the-art.

Besides other common issues of code recommendation, the OOV issue, which is mainly caused by the arbitrariness of identifiers, is considered as one of the most persistent concerns. Several research works tried to address this problem and the open vocabulary approach seems to be a promising one (mentioned in Section 5.2.3). A prediction model with an open vocabulary is supposed to alleviate the OOV issue in the original model (i.e. with a closed vocabulary).

To evaluate how using open vocabulary enhances the original Transformers, or in general, to evaluate whether *CT3* is beneficial for improving code completion models, we performed two steps. Firstly, we implemented a Transformer-based code completion approach in two variants: with a *closed vocabulary model* (i.e. Transformer learns on a fixed set of strings), and with an *open vocabulary model* using BPE as suggested by Karampatsis et al. (2020). In the latter version, each token can be encoded by several subtokens (potentially even letters). Secondly, we evaluated the results using aggregated accuracy and *CT3*-data to observe what additional information we can gain by utilizing *CT3*.

It is essential to indicate that the open vocabulary model uses greedy search for finding the next possible subtoken. We assume that a prediction is correct in this model if all subtokens of

the original token are suggested accurately. This leads to the circumstance that the accuracy after training (which just focuses on the next subtoken in the sequence) is typically higher than the accuracy gained from the evaluation step (reiterated in Section 5.6.3).

5.4 Evaluation

5.4.1 Research Questions

In our evaluation, we addressed the following two research questions:

RQ1: Does the refined evaluation reveal useful information for comparing and characterizing code completion approaches? To answer this question, we conducted an experiment of code completions with a Transformer-based model. We compared two variants of the model, i.e. closed and open vocabulary, and investigate whether the refined evaluation reveals more information about each variant and thus facilitates their comparison. Notably, the findings of this question also respond to our third core research question (CRQ3), specified in Section 1.4 of Chapter 1.

RQ2: Does the open vocabulary model improve the prediction accuracy compared to the closed vocabulary model? The utility of the open vocabulary model is assessed by comparing the completions provided by the two models for each code token and calculating accuracies for each model over all tokens. The answer to this question also verifies the advantage of using the open vocabulary model in addressing the OOV issue.

5.4.2 Experimental Setup

Transformer Adoption

We conducted the experiments using Python150k and JavaScript150k datasets. The model is fitted on the original training sets (i.e. $100k$ each). We used Python 3.7.9, TensorFlow 2.3.0, and an open source Cosy[12] implementing Transformer for code completion.

Instead of translating between two sequences of tokens in different languages as the vanilla Transformer (Vaswani et al., 2017), our task was to predict the most likely next token based on previous tokens. Therefore, the Transformer model used in our experiments features the same architecture as an encoder of the vanilla Transformer, i.e. a stack of identical layers where each layer has two sub-layers.

We employed six layers and six heads like other related works (Kim et al., 2021; Chirkova et al., 2021; Ciniselli et al., 2021b; Ciniselli et al., 2021a). Since these related works used various values for embedding size (e.g. 256, 300, and 384), we chose 300-dimensional embedding, which is the commonly used embedding size of Word2Vec model (Mikolov et al., 2013b). Additionally, we selected 300 as the size of hidden layers (i.e. hidden units).

Our experiments include main and tuning experiments. The former focus on emphasizing the benefits of refined evaluations by using the proposed methodology in comparison of completion

[12]GitLab repository, `https://gitlab.com/Einhornstyle/cosy`, (Accessed: 16 March 2024).

results, obtained from the closed and open vocabulary models. The latter experiments assess effects of some parameters (e.g. token length and window size) on prediction accuracies. To avoid distraction from the main experiments and their results, the tuning experiments are introduced last in the chapter, Section 5.6).

Data Preprocessing

The code token and subtoken sequences are generated by traversing ASTs in Depth-First Search (DFS) order. We eliminated all white spaces, tabs and new lines to reduce noise before collecting tokens for building encoders and creating input files for our Transformer model. Each type or value in the ASTs is represented as an AST node.

Closed and Open Vocabulary Construction

There are special cases that require special values and be reserved in the vocabulary. We utilized `<UNKNOWN>` token to indicate tokens that cannot be encoded by the vocabulary. The token `<PADDING>` is used to fill up the data windows in data files (explained below).

Besides, the token `<ENDTOKEN>` is utilized for marking the end point of a sequence of subtokens and therefore can only be applicable for the open vocabulary case. In other words, if n is the vocabulary size, which is a hyperparameter defined before building the vocabulary, then $n-2$ and $n-3$ are the number of tokens/subtokens in the closed and open vocabularies, respectively.

The **closed vocabulary** is assembled based on the frequency of code tokens in the training dataset. The vocabulary size (e.g. 10,000) determines the number of most frequent tokens being selected for the vocabulary after subtracting the two special tokens as mentioned above. In this way, rare and/or very long tokens hardly contribute to the vocabulary. Hence, a length threshold for building closed vocabulary is not necessary.

For the **open vocabulary** built by BPE approach, we deployed HuggingFace Tokenizers[13]. All tokens in the training dataset are pre-tokenized to build a set of unique words. Frequency of each word is then calculated based on its occurrence in the training set. The open vocabulary first encompasses all characters in the set of unique words and then being expanded by merge rules. Two characters in the vocabulary are combined if their merged character has the highest frequency, compared to the other combinations. The process keeps going until the vocabulary achieves the desired size.

All tokens participate in forming the open vocabulary, which leads to the risk of increasing noise due to very long and rare tokens. To address this issue, we experimented with several thresholds of token length, including unlimited length down to 200, 100, and 50 (detailed in Section 5.6.2). The obtained results show that limiting token length to 50 can create a BPE vocabulary which contains all single letters and a minimum number of missing non-terminal types, which are rarely used.

[13]Fast state-of-the-art tokenizers, https://huggingface.co/docs/tokenizers/python/latest/index.html, (Accessed: 16 March 2024).

Input Data File Creation

To create input data for Transformer models, i.e. sequences of tokens/subtokens, we traversed ASTs in the dataset in DFS order. Similarly to building the vocabulary, rare and/or long tokens are most likely encoded as `<UNKNOWN>` token while using a **closed vocabulary** Consequently, a length threshold is again not needed for this case.

For the **open vocabulary** case, we performed additional experiments on the effect of token length on prediction accuracy (further details in Section 5.6.3). The experimental results show that there should be a threshold for token length when creating data files (e.g. *tfrecord* files if using TensorFlow).

Particularly, there are 0.9% of tokens in the Python dataset that are longer than 30 characters. This applies to 0.3% of tokens in the JavaScript dataset. Together with other reasons (e.g. length distribution, accuracy after training, as presented in Section 5.6.3), we considered 30 as the length threshold for creating our *tfrecord* files.

Challenges of big ASTs. Theoretically, vanilla Transformer models can handle arbitrarily long input sequences. However, in practice, all the input data fed to the model should have the same length with `<PADDING>` tokens added according to the certain lengths of the input sequences (Vaswani et al., 2017). Furthermore, determining the length for the input data involves various factors, such as the length of ASTs and memory capacity. Since ASTs in the datasets are diverse and some can be exceedingly large, it is infeasible to set a value sufficiently high to embed any AST of any length, especially with the limitation of memory.

Sliding window. To integrate large ASTs into the input data, we adapted the same technique as Kim et al. (2021), which splits the large ASTs into smaller chunks with a sliding *window* while keeping the overlaps between windows to preserve the context. Each *window* is defined by a *window_size* (i.e. number of tokens/subtokens within the window) and a *step_size* of the sliding window. To ensure all windows have the same size, the `<PADDING>` tokens are used to pad the last windows of the sequences.

We utilized the same experimental setup as Kim et al. (2021) for our closed vocabulary case with a *window_size* of 1,000 and a *step_size* fixed to 500. However, these values need to be adjusted for the open vocabulary case to prevent dropping contextual information in the *window*, since the sequences of subtokens (i.e. with open vocabulary) can be much longer than the original sequences of code tokens (i.e. with closed vocabulary). After performing several experiments (specified in Section 5.6.4), we selected 2,000 and 1,000 for the *window_size* and *step_size* of the open vocabulary case, respectively.

Experiment Configuration

Table 5.9 presents the settings used for the conducted experiments. The model is trained and evaluated on one GPU (GeForce RTX 2080 Ti or TITAN Xp). The preprocessing and training processes take days[14] while the evaluation for each token prediction with the closed and

[14]It took 20 hours vs. two days for preprocessing and training data on the Python vs. JavaScript datasets with the closed vocabulary, respectively; 1.5 days vs. 4.5 days for the open vocabulary case on the two sets in the same order.

open vocabulary models requires around two weeks (Python dataset) to more than one month (JavaScript dataset) to finish.

Table 5.9: Experiment configuration for *CT3*.

Parameter	Value
`vocabulary_size`	10,000
(`window_size`, `step_size`) for closed vocabulary	(1,000, 500)
(`window_size`, `step_size`) for open vocabulary	(2,000, 1,000)
`batch_size`	8
Number of epochs	10
`max_len_encoder`	50
`max_len_data`	30
Optimizer	Adam

5.4.3 Evaluation Results

Refined versus Aggregated Evaluation

The first line of Table 5.10 compares the aggregated accuracy results of the closed and open vocabulary models on the Python50k dataset. The refined evaluation conducted on this dataset is presented in part (a) of Figures 5.5 to 5.8 for the dimensions syntax type, origin, length, and frequency, respectively. The analysis for the context dimension is similar to other dimensions and omitted in this study.

Notably, in Figures 5.5 to 5.8, we utilized the value *non_leaf* to refer to code tokens that are non-terminal nodes or terminal nodes with token length greater than 30 (due to the length threshold for creating input data files, Section 5.4.2). In part (a) of the mentioned figures, the bars present accuracies while the line shows the relative fraction of a token type over all tokens (i.e. *value_frac*). The pie charts in part (b) of these figures are another representation of the lines in part (a).

Table 5.10: Aggregated accuracy of closed and open vocabulary models.

Dataset	c_acc.	o_acc.	c_oov_o_true*	oov_c^{+}	oov_o^{++}
Python50k	0.68	0.72	$1.35M$	$5.0M$	$0.41M$
JavaScript50k[15]	0.71	0.75	$1.9M$	$6.76M$	$0.4M$

c/o denotes closed versus open vocabulary model, acc. is accuracy.
The accuracies obtained while comparing both models side by side for each token.
* Number of tokens marked as OOV under closed vocabulary model yet correctly predicted by the open one.
$^{+}$ Total number of tokens encoded as OOV with closed vocabulary.
$^{++}$ Total number of tokens encoded as OOV with open vocabulary. In this case, it is also the number of tokens in the dataset that are longer than the threshold of token length (i.e. 30).

[15] Due to time constraints, solely more than 71% of code tokens in the JavaScript50k dataset were assessed. However, together with the results obtained from evaluating the Python50k dataset, this amount of data is still sufficient to confirm the conclusion of our work.

> *Finding 1.* Both metric types show the superiority of the open vocabulary model over the closed one. However, the aggregated metric demonstrates only marginal improvement, while the refined metric reveals significant disparities in certain token types.

The aggregated metric indicates that the open vocabulary model increases the prediction accuracy on Python50k dataset by only 5.88% (first and second columns of Table 5.10). Meanwhile, the refined evaluation clearly reveals that the open vocabulary model outperforms the closed vocabulary model in every dimension of *CT3*, with improvements ranging up to 2.03 times.

One of the reasons that aggregated metrics display only a moderate improvement is the token type *non_leaf* (i.e. internal AST nodes and long token terminal nodes), since it makes up more than half of test instances, but does not benefit much from the open vocabulary model. Detailed advantages of the open vocabulary model over the closed one are analyzed for each dimension as follows:

> *Finding 2.* In terms of **syntax type**, the open vocabulary model exhibits higher accuracy compared to the closed one across most token types. Besides, the refined evaluation unveils greater difficulty in predicting definition token types than usage token types.

Figure 5.5(a) shows that for the dimension syntax type, the open vocabulary model achieves a higher accuracy for all token types, excluding the *exception*, *imp_lib*, and *keyword* types. There is a significant improvement for the *class_usg* type (95.2%). The accuracies of the types *var_usg* and *const_str*, which are two of the most relevant completions in practice, increased by 28.6% and 40.1%, respectively.

Other notable enhancements are achieved for the labels *attribute* (40.2%), *func_usg* (22.5%), and *method_usg* (22.3%). Both closed and open vocabulary models perform quite well for tokens categorized as *keyword* type. However, for this type, the closed vocabulary model slightly outperforms the open vocabulary model by 1.2%. For simplicity, the *unknown* type (0.1% of all tokens) is left out from the bar chart.

The refined evaluation in Figure 5.5(a) also illustrates that it is much harder to predict definition than usage token types. The most impressive evidence for this is the great disparity of accuracy between the *func_def* and *func_usg* types. Their accuracies differ by a factor of 7.4 (closed vocabulary model) and 7.8 (open vocabulary model) in favor of the usage token type. The differences of accuracy between code completion for definition and usage of types related to class, method, and variable are also remarkable.

Apart from the *non-leaf* label in Figure 5.5(b), the top-5 most frequent values of the syntax type dimension consist of *var_usg*, *const_str*, *keyword*, *attribute*, and *method_usg*. For instance, 27.3% of terminal tokens or 9.2% of all code tokens are categorized as the usages of variables, arguments, libraries and function keywords (i.e. *var_usg*). This statistic also emphasizes the importance of predicting already defined identifiers in practice.

In conclusion, the above analysis can inspire the right approaches to improve accuracy, which can not be derived from the aggregated evaluation. The interpretation for the remaining dimensions works similarly.

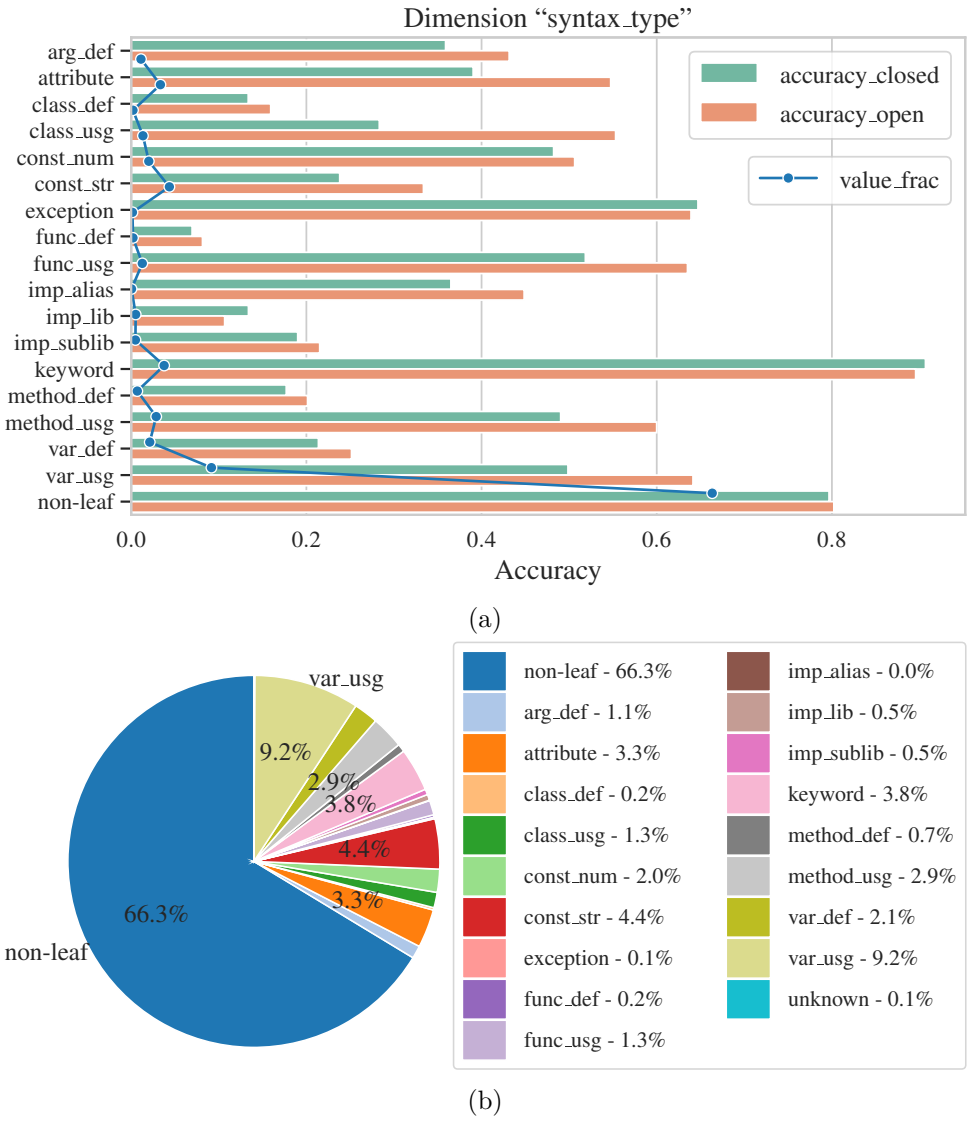
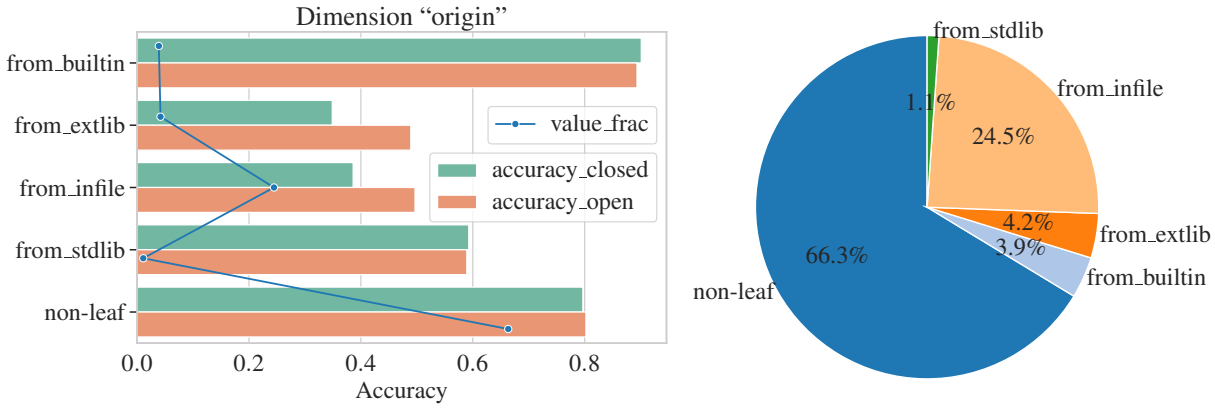


Figure 5.5: Comparing the accuracy of the closed and open vocabulary models for the syntax type dimension (a) and its token types distribution (b).

> *Finding 3.* Concerning token **origin**, the open vocabulary model excels over the closed model in predicting tokens originating from the same file, which constitute the majority of terminal tokens. In addition, suggesting tokens from the same or other source files is harder than suggesting tokens from the built-in and standard libraries of the target language.

Figure 5.6(a) shows that the open vocabulary model outperforms the closed one in most of values of the dimension origin. The closed vocabulary model just slightly increases the accuracy of the open one for values *from_builtin* and *from_stdlib* by 0.8% and 0.6%, respectively. The notable point is the result for the *from_infile* value.

Even though 72.7% of terminal tokens are located in the same file (Figure 5.6(b)), the accuracy of suggesting these tokens is deficient when using the closed vocabulary model (ca. 38.6%). A possible reason for this is the source code file (*from_infile*), as well as the external libraries (*from_extlib*), containing an immense variety of tokens defined by developers, which causes the OOV issue in the closed vocabulary model.



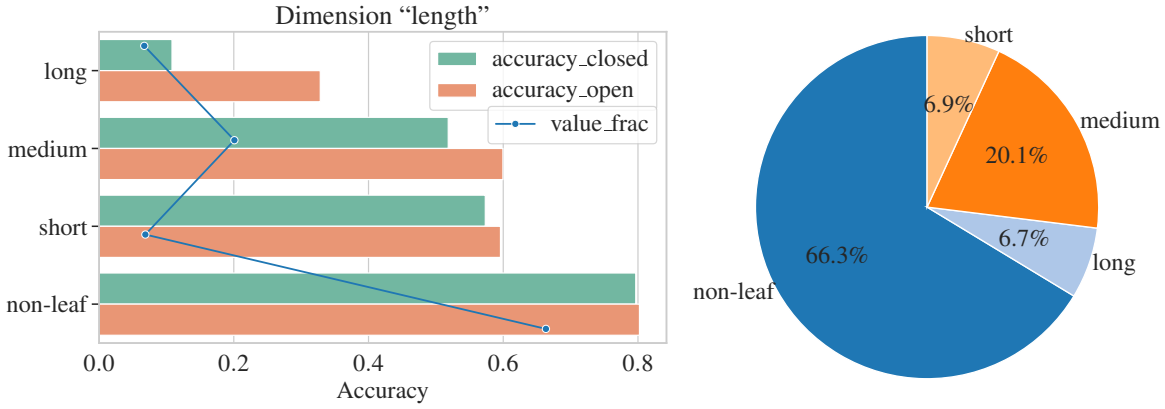
(a) (b)

Figure 5.6: Comparing the accuracy of the closed and open vocabulary models for the origin dimension (a) and its token types distribution (b).

The open vocabulary model improves the prediction accuracies for the labels *from_infile* and *from_extlib* by 28.6% and 40.3%, respectively. However, these accuracies are still lower than the ones for the *from_builtin* and *from_stdlib* labels. This reflects the fact that it is harder to recommend tokens from the same file or from other source files than tokens from the built-in and standard libraries of the target programming language.

> *Finding 4.* On the aspect of token **length**, the open vocabulary model significantly exceeds the rival model on long terminal tokens, despite the modest accuracy of this token type.

The refined evaluation for the length dimension is displayed in Figure 5.7(a). For longer code tokens, the difference between the accuracies of the open and closed vocabulary model increases. Although the overall accuracy for *long* tokens is not high (ca. 32.8%), the improvement by a factor of 3.03 is still impressive. This finding is crucial, as *long* code tokens are likely to be requested for code completion (Hellendoorn et al., 2019).



(a) (b)

Figure 5.7: Comparing the accuracy of the closed and open vocabulary models for the length dimension (a) and its token types distribution (b).

The tokens tagged with the *medium* label make up the largest fraction of terminal tokens in this dimension (Figure 5.7(b)). In this case, the closed vocabulary model is outperformed by the

open vocabulary model with a margin of 15.7%. For *short* code tokens, the accuracy difference is not substantial.

> *Finding 5.* The open vocabulary model consistently outshines its competitor in token **frequency** across all labels. The analysis results also reaffirm the difficulty of predicting low frequency tokens.

The open vocabulary model also surpasses the closed one for all values of the frequency dimension (Figure 5.8(a)). The *low* frequency tokens not only constitute a notable fraction of the terminal tokens (Figure 5.8(b)), but are also quite difficult to predict when using the closed vocabulary model. The increased accuracy for these tokens (44.1%), together with the above analysis results, emphasizes the advantage of using the open vocabulary model instead of the traditional closed one.

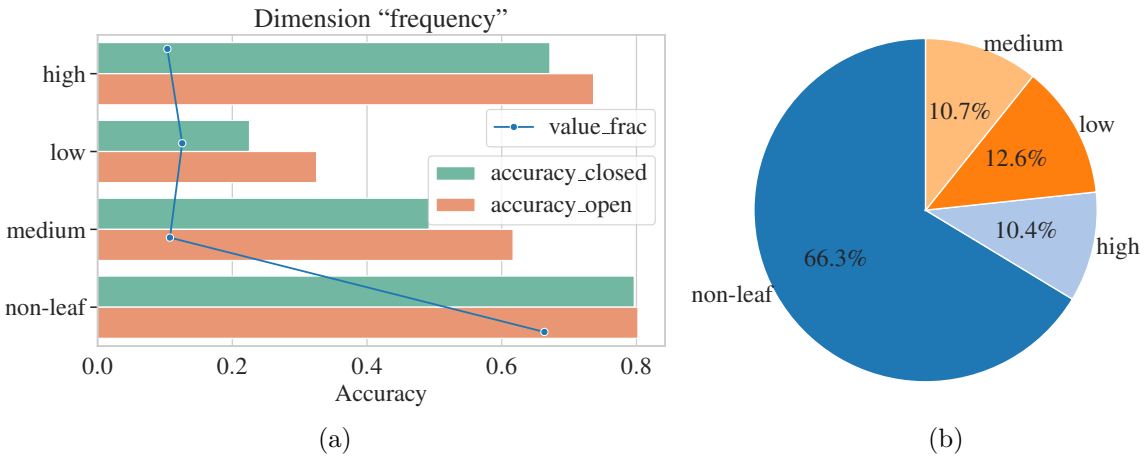


Figure 5.8: Comparing the accuracy of the closed and open vocabulary models for the frequency dimension (a) and its token types distribution (b).

> *Finding 6.* Results of the refined evaluation also highlight that predicting non-terminal and lengthy tokens significantly influences the aggregated accuracy, diminishing the advantage of the open vocabulary model when using this conventional metric.

As indicated in parts (b) of Figure 5.5 to 5.8, the *non-leaf* value makes up the majority of tokens in the dataset (66.3%), which leads to the challenge that improving accuracy in other values does not have much effect on the aggregated results. In other words, the prediction of non-terminal and very long tokens dominates the aggregated accuracy and obscure the preeminence of the open vocabulary model.

The thorough comparison between traditional aggregated and refined metrics in evaluating prediction outcomes yields valuable insights into addressing the first research question introduced in Section 5.4.1 of this chapter.

> **A-RQ1. The refined evaluation exceeds its competitor in providing detailed information on completion models, guiding further improvements.**

Open versus Closed Vocabulary

> *Finding 7.* The open vocabulary model outclasses the closed vocabulary model in terms of performance across most dimensions and token types, while mitigating the OOV issue.

As analyzed above, the charts in Figure 5.5 to 5.8 reveal that the open vocabulary model enhances the prediction accuracy of the Transformer-based model across most dimensions and values of *CT3*. Instances where the closed vocabulary model performs slightly better than the open one exhibit insignificant differences (0.6% to 1.2%), except for the label *imp_lib* (20.3%), which captures only 0.5% tokens of the whole dataset or 1.5% of terminal tokens.

The difference in accuracies between the two models is particularly impressive for the usages of identifiers (e.g. classes or variables), tokens from in-file or external libraries, and low frequency long tokens. These token types are closely related to the OOV issue, which is caused by the arbitrariness of identifiers.

The refined evaluation results confirm the expectation that the open vocabulary model can alleviate the OOV issue in the traditional approaches. To complement this point, we counted the number of OOV tokens encountered when using the closed vocabulary model and determined how many of these tokens can still be recommended correctly by the open vocabulary model. We also calculated the number of tokens encoded as OOV with the open vocabulary to compare with the statistic of the closed one.

It is worth noting that, in the open vocabulary scenario, the count of OOV tokens is equivalent to the number of *excluded tokens*, i.e. those exceeding the length threshold. This is due to the absence of tokens shorter than the length threshold encoded as OOV with the open vocabulary. The last three columns of Table 5.10 presents our calculation.

The open vocabulary model accurately predicts 27.0% and 28.1% of OOV tokens in the Python50k and JavaScript50k datasets, respectively. Moreover, the open vocabulary decreases the number of tokens encoded as OOV with the closed vocabulary by factors of 12.2 and 16.9 for the Python and JavaScript datasets, respectively. This again confirms the advantage of the open vocabulary approach.

The above analysis also answers the second research question specified in Section 5.4.1.

> **A-RQ2. The open vocabulary model demonstrates superiority in predicting identifier usages, tokens from local files or external libraries, and infrequent long tokens, compared to the closed model.**

5.5 Discussion

This section outlines challenges encountered in developing the *CT3* approach, threats to validity, and reaffirms our response to the third core research question (CRQ3), introduced in Section 1.4 of Chapter 1.

5.5.1 CT3 Challenges

The first and most important step of constructing *CT3* methodology is to determine dimensions and possible values for each dimension. In this subsection, we discuss two challenges occurred while conducting the *CT3* schema for Python.

Identifying differences among token usages. Considering the syntax type dimension, the main purpose is to separate between various token types, as well as their definitions and usages (e.g. definition and usage of function and method). While the prediction of token definitions is harder to achieve, it is also less crucial than the prediction of token usages. In other words, distinguishing between the usages of different token types is important.

However, for some token types, it is hard to accomplish this distinction. One example for this is the usages of imported libraries, defined variables and declared arguments all being treated as variable usages in Python. In these cases, the analysis can be simplified based on interests of developers. For instance, a token can be considered as an imported library usage if it is identified as a *var_usg* in the syntax type dimension and its origin value is *from_extlib* or *from_stdlib*.

Determining definitions of local variables. We assumed that local variables are defined by assignments in ASTs and only the first assignment of each variable within a scope (i.e. file, class, method or function) is captured. The definition of local variables inside `lamda` functions, `if`/`else` blocks or loops might be obtained by slightly modifying the AST parser.

Overcoming these obstacles will optimize the *CT3* schema, which is beyond the scope of our work, and therefore will be investigated in future work. Besides, the effect of these challenges on the evaluation result is negligible since the current schema still provides fundamental and essential dimensions of code token types (e.g. definitions and usages of identifiers, token length, and token origin).

Moreover, analyzing other token types based on the ASTs is straightforward and can be applied to other programming languages with minimum effort (e.g. adjusting the types of nodes in ASTs between languages). We already published the scripts for conducting *CT3* schema for Python, which support user adding/modifying dimensions and values in each dimension easily.

5.5.2 Threats to Validity

There are several threats to validity of our work. We analyze them as follows:

Training corpus. We employed the original train-test split of Python150k and JavaScript150k datasets to easily compare with the state-of-the-art. However, Chirkova et al. (2021) suggested that the datasets should be redistributed and deduplicated to avoid data leaks. We also observed that the datasets are quite noisy with a lot of empty or long tokens and auto-generated files. These characteristics might affect the semantics of the prediction. It also underlines a need for a set of standardized benchmarks for code completion.

Language specificity. The utility of *CT3* methodology is only evaluated for Python. The code token types are obtained by analyzing ASTs and the relationship between tokens, which is slightly different for each programming language. However, the dimensions of *CT3* are determined by the demands of developers for code completion in practice.

Particularly, predicting identifiers over punctuation and distinguishing between definitions and usages of identifiers motivate the syntax type dimension, while saving typing effort prompts the length dimension. Perceiving the importance of local context/information in the predictions derives origin and context dimensions. Ultimately, suggesting rare and difficult completions as vital cases for real world efficacy inspires the frequency dimension.

These interests are applied for every programming language. Furthermore, most of the values identified for the dimensions of *CT3* are critical elements, which are shared among programming languages. Accordingly, we expect that our results will generalize to other languages.

Model comparison. We selected a Transformer-based model in two variants (i.e. with open and with closed vocabulary) as representative ML-based models for this work due to the dominance of Transformers in comparison to other models. Besides, Transformers also achieve notable results on other tasks, such as classification (Kanade et al., 2020), vulnerability/code clone detection (Ding et al., 2021), and Natural Language to code (Ahmad et al., 2021) with leading tools GitHub Copilot and AlphaCode[16].

With the rapid development of Transformers, it is unlikely that previously known models can surpass this model. We assume that a comparison between other approaches (e.g. Transformers versus GNN) using *CT3* would further confirm the advantages of our proposed methodology.

Out-of-Vocabulary (OOV) issue. Handling the OOV issue by BPE is not the main focus in this work and is only used to demonstrate the proficiency of *CT3* in supporting a refined evaluation. In the paper of Chirkova et al. (2020), the authors mention that splitting tokens into subtokens makes it hard to apply structure-aware models. But our results show that the open vocabulary model constructed with BPE outperforms the closed one in every category of token types. However, the efficiency of solutions for the OOV issue needs further investigation and is out of scope of this work.

5.5.3 Response to CRQ3

The third core research question (CRQ3), outlined in Chapter 1, Section 1.4, targets to compare traditional aggregated evaluation methods with refined strategies in revealing insights into code completion approaches. The extensive analysis with the first six findings summarized in Section 5.4.3, alongside the discussion above clarify our answer for this matter.

> **A-CRQ3. Traditional aggregated evaluation lacks depth in comparing code completion model performance, whereas the refined evaluation addresses this deficiency, offering valuable viewpoints for potential advancements.**

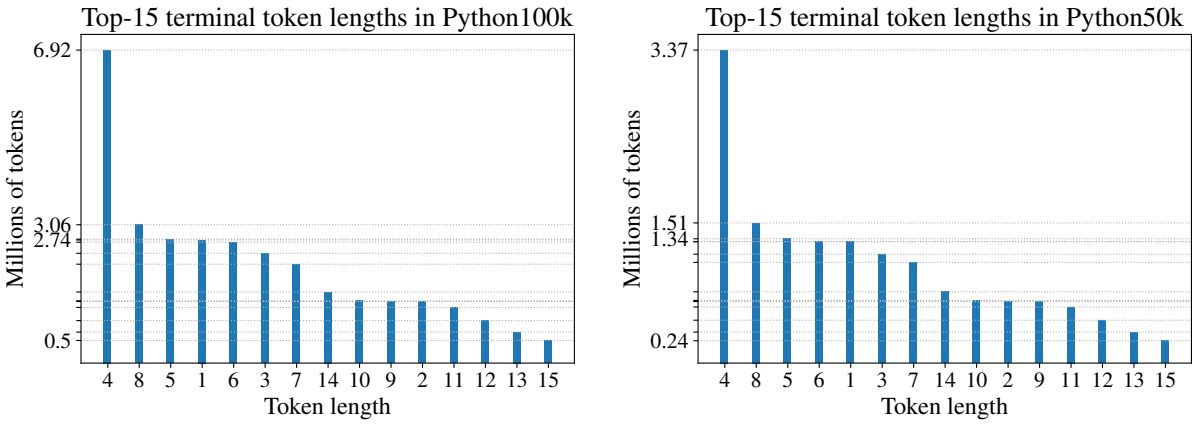
5.6 Auxiliary Experiments

This section presents our additional statistics and results of tuning experiments to clarify some values and thresholds chosen for the main experiments described above.

[16] AlphaCode attention visualization, https://alphacode.deepmind.com/, (Accessed: 17 March 2024).

5.6.1 Length Distribution of Terminal Tokens in Python150k

While preprocessing the code tokens in the dataset, we recorded their length distribution to identify thresholds for the three values in the length dimension of the *CT3* schema used for Python (i.e. *short*, *medium*, and *long*). Figure 5.9 displays the top-15 common terminal token lengths in the Python150k dataset.



(a) – Approx. $35M$ terminal tokens in total

(b) – Approx. $17M$ terminal tokens in total

Figure 5.9: Top-15 common terminal token lengths in the Python150k dataset for the training 100k dataset (a) and the evaluation 50k dataset (b).

Approximately 19.7% of terminal tokens in both training and evaluation datasets are 4 characters long, with longer lengths representing a small fraction. Figure 5.9 illustrates that the majority of tokens range between 4 and 10 characters. Moreover, the training and evaluation datasets contain nearly $4k$ and $3k$ different lengths, respectively. Consequently, longer terminal tokens are less frequent in the dataset, reducing their predictability.

Moreover, since typing effort plays an important role in the code completion task (Hellendoorn et al., 2019), we preferred to focus on the code tokens that can help with (i) saving typing effort as well as (ii) covering a large part of the dataset. Based on these criteria and the analysis above, we chose 4 and 10 as the thresholds for the length dimension of *CT3* schema (for Python). Figure 5.7(b) emphasizes that medium-length tokens (i.e. $4 \leq$ `token_length` ≤ 10) indeed account for nearly 60% of the terminal tokens in the dataset.

5.6.2 Length Threshold for Open Vocabulary Building

As mentioned in Section 5.4.2, building an open vocabulary can be disturbed by very long tokens. For this reason, we investigated several values of token length while using HuggingFace Tokenizers to build the open vocabulary on the Python100k dataset (i.e. training set).

Selection Criteria

Table 5.11 summarizes the obtained results under three criteria: (i) the presence of single letters in the vocabulary (second column), (ii) number of missing non-terminal types (third column),

and (iii) how the code tokens are encoded by considering the maximum length, most frequent length, and average length of generated sequences of subtokens, and the number of *included tokens* encoded as OOV in the training dataset (the last four columns of the table, respectively). Here, *included tokens* are those with lengths below the threshold.

Table 5.11: Exploration of length thresholds for Python100k dataset's open vocabulary building.

Max length	Presence of single letters	No. of absent non-terminal types*	Max seq. length of subtokens	Most freq. seq. length of subtokens	Average seq. length of subtokens	No. of included OOV tokens**
unlimited	✓	67	193	2	2.44	0
200	✓	64	193	2	2.43	0
100	✓	63	96	2	2.37	0
50	✓	63	30	2	2.24	0

* The total number of non-terminal types is 154.
** Tokens that satisfy the length threshold but are encoded as OOV.

Firstly, since each code token is encoded by several subtokens with the main purpose of addressing the OOV issue, the learned vocabulary should contain all single letters[17] (in both uppercase and lowercase) to make sure that code tokens can possibly be encoded.

Secondly, non-terminal tokens in the training dataset are included while building the vocabulary and training the completion model. In addition, types of non-terminal tokens are more frequent than values of terminal tokens. As a result, we expected that the learned vocabulary also covers these non-terminal types at an acceptable rate.

Ultimately, we assumed that a code token is predicted properly in the open vocabulary case if all of its subtokens are suggested correctly. Consequently, the shorter the sequence of subtokens, the higher the probability for a correct prediction. Moreover, the learned vocabulary should reduce the number of OOV tokens in the dataset after encoding.

Considered Length Thresholds

For Python100k dataset. Since our main purpose in this work is to propose the methodology for a refined evaluation and to illustrate the utility of the proposed approach, finding optimal values for all the thresholds is beyond this work. Therefore, we only experimented with token lengths from unlimited down to 50, which is higher than the threshold for creating data files in Section 5.6.3 to make sure that long tokens have less subtokens.

While the open vocabulary learned by all the thresholds covers single letters and produces zero *included tokens* as OOV, limiting the length to 50 brings acceptable results with 63 non-terminal types missing, alongside the smallest maximum and average lengths of subtoken sequences, compared to other thresholds in Table 5.11. Besides, we examined the missing types of non-terminal tokens and most of them are rarely used, e.g. `CompareLtLtLt`, `CompareLtLtEEqLtLtLt`, or `AugAssignLShift`.

For JavaScript100k dataset. We assume that the token lengths in the JavaScript dataset follow the same trend to the ones in the Python set. Hence, we analyzed the JavaScript100k

[17] In our experiments, we mostly focused on English letters.

dataset using the above criteria while limiting the token lengths to 50. The result verifies our inference through the obtained vocabulary with only one missing non-terminal type, the maximum length of subtoken sequences of 31 and an average token length of 2.17. As expected, there are no *included tokens* encoded as OOV while using the built open vocabulary. Accordingly, we chose 50 as the length threshold for building the open vocabulary in both datasets.

5.6.3 Length Threshold for Input Data File Creation in Open Vocabulary Case

Token Length Distribution

For the code completion task, the quality of input data is an essential factor, since very long tokens might increase noise while training the models. To minimize this risk, we eliminated very long tokens from the dataset before creating data files and considered them as `<UNKNOWN>` tokens after the preprocessing step. We again studied the length distribution of code tokens in Python and JavaScript datasets considering both non-terminal[18] and terminal tokens to identify the threshold for very long tokens.

Figure 5.10 presents the top-15 of the most common token lengths in the Python and JavaScript datasets for both training and evaluation sets. Most code tokens in the Python150k dataset have a length of 4 characters and the tokens with lengths between 4 and 10 make up more than 60% of the dataset. The thresholds 4 and 14 can be applied similarly to the JavaScript150k dataset where the most common length is 10. In other words, code tokens longer than 10 for Python and 14 for JavaScript occur less and less in the dataset.

Considered Length Thresholds

For Python150k dataset. To determine the acceptable-longest-length for the code tokens in our experiments, we tested several lengths from 200 down to 30 for code tokens in the Python150k dataset first and got the results in Table 5.12. If the allowed maximum length of code tokens is 200, only 0.1% of code tokens have a length above the threshold. Similarly, reducing the threshold down to 100 and 30 brings 0.2% and 0.9% of code tokens that longer than the max length, respectively.

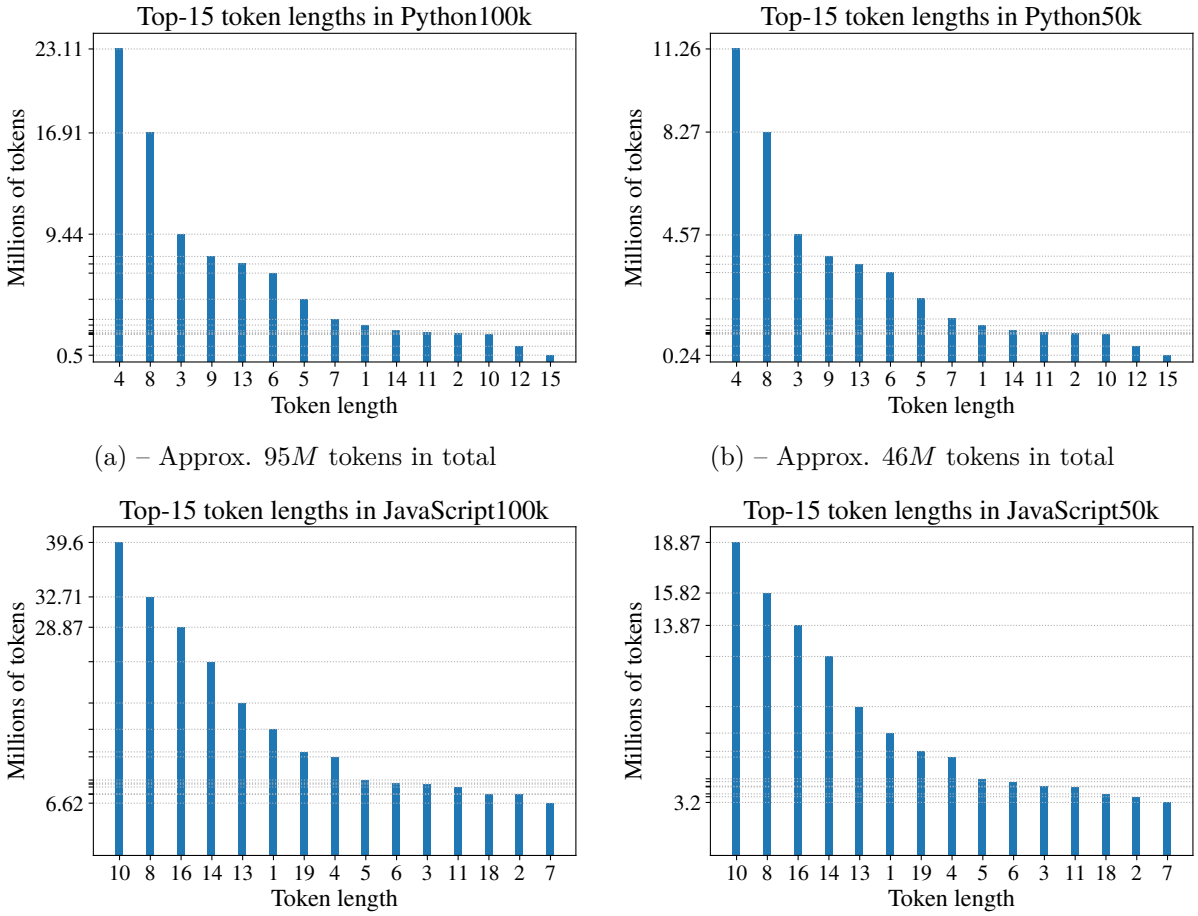
Table 5.12: Investigation of length thresholds for Python150k dataset's input data file creation.

Max length	Portion of *excluded tokens**	Accuracy after training**
200	0.1%	0.8179
100	0.2%	0.8255
30	0.9%	0.8363

* *Excluded tokens* are tokens having lengths longer than the max length.
** The model was trained on Python100k dataset for 10 epochs.

However, after training the model for 10 epochs, limiting the max length to 30 achieves the highest accuracy for the 10th epoch, among other values. This means the model might be

[18]Non-terminal tokens are also fed to the completion models.



(a) – Approx. $95M$ tokens in total

(b) – Approx. $46M$ tokens in total

(c) – Approx. $269M$ tokens in total

(d) – Approx. $128M$ tokens in total

Figure 5.10: Top-15 common token lengths in the Python150k dataset (a) – (b) and the JavaScript150k dataset (c) – (d).

affected by noise created from the long tokens. Besides, discarding 0.9% of code tokens by excluding them from the dataset does not affect the evaluation much. Therefore, we chose 30 as the length threshold for creating data files in the Python dataset.

For JavaScript150k dataset. Since the most common length of tokens in the JavaScript150k dataset is longer than the one in the Python150k dataset, we predicted the threshold for the JavaScript dataset to be higher. However, only 0.3% of code tokens in the JavaScript150k dataset have a length longer than 30. Furthermore, in this case the accuracy after training is 0.8642, which is sufficiently high. Accordingly, we still selected 30 as the length threshold for creating data files in the JavaScript150k dataset.

Here, the post-training accuracy, notably in the open vocabulary case, surpasses that attained during evaluation. Namely, with a fixed length threshold of 30, training on the Python100k dataset yields an accuracy of 0.83, which decreases to 0.72 during evaluation, as depicted in Table 5.10. This results from TensorFlow Keras achieves the former accuracy by solely considering the correctness of the next subtoken prediction during training, whereas the latter accuracy employs a stricter rule, i.e. all subtokens of the original token to be predicted correctly.

5.6.4 Window Size for Input Data File Creation in Open Vocabulary Case

Selection Criteria

As mentioned in Section 5.4.2, we chose 1,000 and 500 as the *window_size* and *step_size*, respectively, for creating input data with the closed vocabulary in both the Python and JavaScript datasets. Table 5.13 presents our experimental results to determine these two values for the open vocabulary case, taking into account the following factors:

- The value of *step_size* is inferred as half of *window_size* for the sake of simplicity.
- Each token is encoded to at least two subtokens (i.e. the token itself and the mark symbol `<ENDTOKEN>`). As a result, to prevent omitting information, the *window* used to incorporate sequences of subtokens (i.e. with open vocabulary) should be larger than the one used for sequences of tokens (i.e. with closed vocabulary).
- The *window_size* value needs to be constrained due to the limited capacity of our GPUs.
- *batch_size*, i.e. the number of data windows grouped in a batch, also affects the training process. We established a *batch_size* threshold for each considered size of the *window*, since values higher than the threshold exceed the capacity of our GPUs.
- Another important element is the Estimated Time of Arrival (ETA), which is the estimated time that the model needs to complete one *epoch* (i.e. one training iteration). In other words, the training time can be measured by multiplying ETA with the number of epochs. We expected our model to be trained in an acceptable duration of one to two days.
- Ultimately, the accuracy obtained after training should be sufficiently high in an adequate amount of time (e.g. two days). Besides, as mentioned at the end of Section 5.6.3, the accuracy declines in the evaluation step since all subtokens of a token must be predicted correctly to form a proper prediction in the open vocabulary case.

Considered Window Sizes

For Python150k dataset. We investigated three different sizes for the *window* (i.e. $5k$, $3k$, and $2k$) with corresponding values of *epoch* and *batch_size*. Table 5.13 shows that setting *window_size* to $5k$ makes training per *epoch* faster than the case of *window_size* $3k$ due to the reduced *batch_size*. However, the accuracy after training of both sizes $5k$ and $3k$ are just half of the one obtained by the *window_size* $2k$ with the same number of epochs.

For the size of $3k$, the accuracy improves after performing 10 more epochs but still lower than the one from size $2k$. Moreover, the ETA values for the *window_size* $3k$ are pretty high. Consequently, $2k$ is the most suitable size for the *window* based on the above criteria and in comparison to the other values. The *step_size* is then adjusted to $1k$ accordingly.

For JavaScript150k dataset. We applied the same value to the *window_size* used for the open vocabulary in the JavaScript150k dataset, since the maximum, most frequent, and average

Table 5.13: Examination of window sizes for Python150k dataset's input data file creation.

Window size	No. of epochs	Batch size*	Acc. after training	ETA (h:mm)
$5k$	10	1	0.4468	5:18
$3k$	10	4	0.4469	6:48
$3k$	20	4	0.5243	8:01
$2k$	10	8	0.8363	3:54

* The highest value of *batch_size* that our GPUs can handle.

lengths of sequences of subtokens in both datasets are considerably close (mentioned in Section 5.6.2) and they have the same setup for the closed vocabulary case.

5.7 Summary

This chapter introduces our proposed methodology called *Code Token Type Taxonomy (CT3)* for a refined evaluation and comparison of code completion models. Our empirical study shows that *CT3* effectively characterizes and compares the accuracy of various approaches. Besides, we found that the open vocabulary method notably improves the accuracy of Transformer-based models in code completion tasks, especially regarding the usage of defined variables and literals.

We also compared the state-of-the-art of ML-based code completion approaches. The overview shows that there is a demand for a set of standardized benchmarks for a rigid and reproducible comparison of prediction models. We published the *CT3* information[19] and the analyzer source code[20] for the Python150k dataset. Potential further work includes extending the *CT3* method to other programming languages and datasets, as well as implementing specialized code predictors according to the proposed *CT3* schema.

The proposed methodology also addresses the *programming barrier* </>, *reuse problem*, *and scalability problem* outlined in Chapter 1, Section 1.1. Particularly, our refined evaluation highlights its advantages in offering insights to improve code completion models, tackling the *programming barrier* </>. Moreover, our published source code and data enable examination of other code completion approaches, mitigating the *reuse problem*. Ultimately, the parallel implementation of *CT3* functions partially resolves the *scalability problem*.

Additionally, the comprehensive analyses obtained from the refined evaluation respond to our third core research question (CRQ3), specified in Section 1.4 of Chapter 1. We anticipate that employing this refined evaluation will yield in-depth examinations on the benefits introduced by individual components within ensembles of code completion models, such as the *Extended Network* presented in Chapter 4.

In the upcoming chapter, our focus shifts from suggesting next code tokens, as featured in this and the preceding chapters, to simplifying programming tasks with NLP techniques. Our next approach involves translating NL into code snippets, while enhancing the interpretability of ML model behaviors.

[19] Code token type data, https://doi.org/10.5281/zenodo.5733013.
[20] GitLab repository, https://gitlab.com/pvs-hd/published-code/code-token-type-taxonomy. (Accessed: 18 March 2024).

One-shot Correction

CHAPTER 6

Enhancing Code Generation Models through User Feedback and Decomposition Techniques

For our final research contribution, we targeted to improve Artifical Intelligence (AI) models for code generation. Exemplified by GitHub Copilot and TabNine, code generation continues to be an integral feature of modern Integrated Development Environments (IDEs) and receives significant attention. This feature operates at various levels such as generating next code tokens, completing functions, and converting Natural Language (NL) to code. However, code generation may shift code writing tasks towards code reviewing, which involves modification from users.

Nevertheless, despite the advantages of feedback from users, their responses remain transient and lack persistence across interaction sessions. This stems from the inherent characteristics of generative AI models, which require explicit re-training for new data integration. Moreover, the non-deterministic and unpredictable nature of AI-powered models limits thorough examination of their unforeseen behaviors.

We proposed a methodology named *One-shot Correction* to mitigate these issues in NL to code translation models with no extra re-training. We utilized decomposition techniques to break down code translation into sub-problems. The final code is constructed using code snippets of each query chunk, extracted from user feedback or derived from a generative model as needed.

Our evaluation indicates comparable or improved performance compared to other models. Furthermore, the methodology offers straightforward and interpretable approaches, which enable in-depth analysis of unexpected results and facilitate insights for potential enhancements. We also illustrated that user feedback can substantially improve code translation models without re-training. Ultimately, we developed a preliminary Graphical User Interface (GUI) application to demonstrate the utility of our methodology in simplifying customization and assessment of suggested code for users.

Section 6.1 outlines our motivation and contributions in this work, while Section 6.2 provides an overview of the background and related work. Our methodology is described in Section 6.3. Section 6.4 presents our experiments in detail, followed by the evaluation results being analyzed in Section 6.5. We discuss threats to validity, potential enhancements, and our answer to the last core research question (CRQ4) in Section 6.6. Additionally, our GUI application is introduced in Section 6.7. Finally, we conclude the chapter in Section 6.8.

This chapter is based on our peer-reviewed publication (Le et al., 2024).

6.1 Introduction

The widespread adoption of generative AI models has facilitated NL-related tasks (Gozalo-Brizuela et al., 2023). Consequently, the integration of NL-to-Code translation has become a sought-after feature in many code-centric tools (Xu et al., 2022). Notable examples include GitHub Copilot[1], TabNine[2], Amazon CodeWhisperer[3] and ChatGPT[4].

Benefits and shortcomings of generative AI models. There are divergent opinions regarding the advantages and security of these AI tools, specifically GitHub Copilot. Various studies and surveys have shown the benefits of Copilot in assisting developers (Bird et al., 2022; Vaithilingam et al., 2022; Dakhel et al., 2023). However, other empirical studies (Imai, 2022) reveal that Copilot results in higher productivity but with lower quality of code.

Particularly, concerns have been raised about Copilot recommending code that relies on non-existing helper functions or undefined variables (Nguyen et al., 2022). Several studies focus on the vulnerability and security of code generation tools (Pearce et al., 2022; Asare et al., 2022), as well as the uncertainty surrounding the licensing of generated code (Bird et al., 2022).

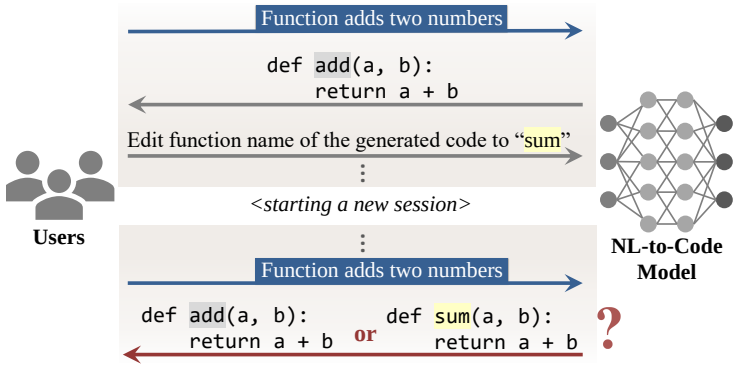
Notably, a study of Bird et al. (2022) reveals that developers devote more time to reviewing AI-generated code than writing it. This emphasizes the demand of aiding developers in better understanding and evaluating the generated code, which is constrained by the unpredictable behavior of AI models. Furthermore, the validation of the code's origin is essential, but currently limited to a single IDE development session.

Omission of user feedback. Although generative AI tools have advanced rapidly, users do not always obtain the desired code outcomes. One of the primary factors is the quality of the prompt, which involves NL features such as implication, association, and ambiguity (Reynolds et al., 2021). Interactive programming has captured considerable attention as one of the prominent approaches to tackle these issues of NL (Shin et al., 2021; Heyman et al., 2021; Schlegel et al., 2019; Elgohary et al., 2021; Su et al., 2018; Cai et al., 2023).

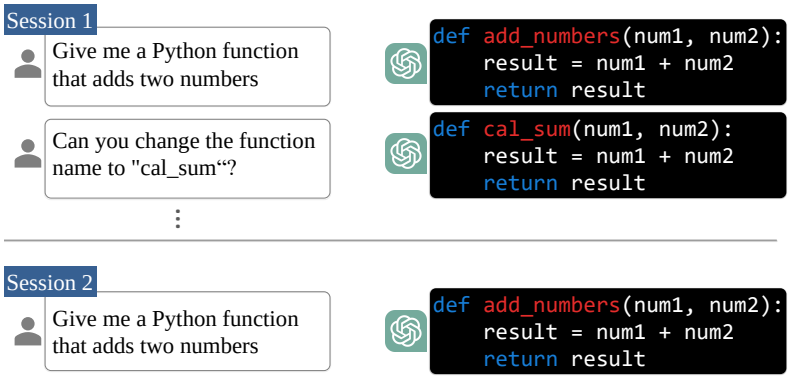
This concept encompasses users engaging with models in an iterative manner through *low-code* approaches, until they attain the expected result. However, despite the leverage of user feedback, its persistence is confined to a single conversational session (Bird et al., 2022). This limitation arises from the inherent properties of generative AI models, which require explicit re-training to integrate new data or feedback from users.

Figure 6.1 illustrates a simple scenario of interactive programming, underlining the issue of recalling cross-session user feedback in a current generative AI model. In Figure 6.1(a), users initially submit the NL query "Function adds two numbers" and receive a Python code representing the `add` function. Subsequently, users request to rename the function from `add` to `sum` and proceed with further unrelated queries. After several inquiries in a new session, users once again input the same query, "Function adds two numbers". The question arises as whether the model should return to users the function with the name `add` or `sum`.

[1] AI developer tool, https://github.com/features/copilot.
[2] AI coding assistant, https://www.tabnine.com/.
[3] AI-powered productivity tool, https://aws.amazon.com/codewhisperer/.
[4] AI chatbot, https://openai.com/blog/chatgpt.
(Accessed: 19 March 2024).



(a) General scenario



(b) Applying the scenario to ChatGPT (GPT-3.5)

Figure 6.1: A simple scenario of interactive programming, highlighting the issue of utilizing user feedback across sessions.

We applied this scenario to ChatGPT, one of the recent prominent tools, and obtained the results as illustrated in Figure 6.1(b). Even though the user has corrected the function name in *Session 1* (i.e. `add_numbers` to `cal_sum`), ChatGPT still returns the original code snippet (i.e. function named `add_numbers`) in *Session 2*. In practice, the repetitive user modifications (e.g. renaming, restructuring) of generated code can be inefficient and frustrating.

Contributions of this work. To address the aforementioned challenges, we proposed a methodology to develop a *user-feedback-driven* NL-to-Code model. This method requires no additional re-training. We aimed to provide interpretable, straightforward approaches to enable comprehension of code provenance and facilitate thorough analysis of unexpected results. Our contributions in this work are as follows:

- *A One-shot Correction methodology.* We introduced an approach to integrate user feedback into generative AI models without re-training while supporting intensive inspection of incorrect outcomes. An additional memory for user feedback and k-Nearest Neighbor (KNN) methods are employed to accumulate and retrieve correction information across sessions. To tackle the code's origin issue, we adopted the techniques from decomposition in problem solving. Each natural language query is divided into segments and the final code snippet is constructed from sub-snippets attained for each segment. The NL-to-code translation of each query chunk is performed through either the additional memory or a generative AI model.

- *A prototype and an extensive comparison.* To illustrate the utility of our methodology, we deployed a prototype of *One-shot Correction* based on GPT-3.5-Turbo-0301 model[5] and conduct an extensive comparison between the code generated by GPT-3.5-Turbo-0301 and our prototype. The evaluation results justify the concept of our methodology and provide insights into the behavior of all models when combined with user feedback.
- *A Graphical User Interface application.* We developed a preliminary GUI to exhibit the benefit of using *One-shot Correction* in customizing and interpreting the generated code snippets. Users can convert an NL query to a Python function, modify identifier names in the produced code snippet, and preserve the correction information for future reference.
- *Source code and data.* To facilitate reproducibility and reuse of our methodology, we published our source code, along with test suites and evaluation results[6].

Notably, we adopted the *CT3* schema from Chapter 5 and tailored it for this study. Similar to Chapter 5, this chapter primarily targets the *programming barrier* </> and partially addresses the *reuse problem* ♻ through our published source code and data. Additionally, the evaluation results obtained from our methodology answer to the final core research question (CRQ4), defined in Chapter 1, Section 1.4.

6.2 Background and Related Work

6.2.1 Generative Artificial Intelligence for Code

Generative AI models are capable of producing novel content across various formats, including text, image, video, or audio (Gozalo-Brizuela et al., 2023). In the context of code generation, generative AI leverages the *naturalness hypothesis* (Hindle et al., 2016; Sun et al., 2022; Weisz et al., 2022), which posits that software can be viewed as a form of human communication (further details in Section 2.2.2 of Chapter 2). Consequently, techniques applicable to NL can also be employed for code generation.

Various approaches, spanning from probabilistic (Bielik et al., 2016; Li et al., 2018) to Machine Learning (Kim et al., 2021; Svyatkovskiy et al., 2020), or ensembles of models (e.g. the *Extended Network* in Chapter 4), have been proposed to validate this hypothesis. Transformer (Vaswani et al., 2017) has emerged as the dominant architecture, serving as the foundation for notable models like PLBART (Ahmad et al., 2021), CodeBERT (Feng et al., 2020), Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), and GPT-3.5[5].

These models support a wide range of code-related tasks, including code summarization, code translation across programming languages (Ahmad et al., 2021), code documentation generation (Feng et al., 2020), code auto-completion based on comments or existing code (Chen et al., 2021; Ahmad et al., 2021), and are even able to challenge humans in programming competitions (Li et al., 2022).

[5] Models from OpenAI, https://platform.openai.com/docs/models/gpt-3-5.
[6] GitLab repository, https://gitlab.com/pvs-hd/published-code/one-shot-correction-ase. (Accessed: 19 March 2024).

Several methods have been proposed to enhance generative AI models. These approaches involve expanding input queries with contextual information, such as code token types (Izadi et al., 2022), preceding task queries and code outputs (Nijkamp et al., 2022). Other methods entail integrating a supplemental retriever (Lu et al., 2022; Parvez et al., 2021) or incorporating an additional memory component (Wu et al., 2022; Fan et al., 2021; Khandelwal et al., 2020).

However, these approaches either require training or overlook the potential of leveraging user feedback as a valuable resource. Moreover, the non-deterministic and unpredictable nature of the underlying AI model restricts in-depth analysis of unexpected behaviors (Bird et al., 2022).

6.2.2 Interactive Programming

The quality of the NL prompt significantly impacts the accuracy of NL translation models (Reynolds et al., 2021). Various techniques have been proposed to address the ambiguity of NL and bridge the gap between NL and programming languages. These methods include heuristic methods, semantic parsing (Shin et al., 2021), and interactive programming. The latter has gained notable attention as a prominent approach (Heyman et al., 2021).

Methods supporting user interaction comprise binary validation of target results (Iyer et al., 2017), multiple-choice questions (Gür et al., 2018), selection from a list of options (Schlegel et al., 2019), decomposition and modification of sub-components using predefined values (Su et al., 2018), feedback through NL queries (Elgohary et al., 2021), and workflow updates instead of direct result modification (Cai et al., 2023). Although user feedback has been shown to be advantageous in these studies, its persistence is limited to a single interaction session.

6.2.3 Decomposition in Problem Solving

Our methodology draws inspiration from a widely known heuristic in problem solving, which involves the decomposition of the problem into manageable sub-problems (Egidi, 2006). This approach is valuable in software development (Charitsis et al., 2022), and particularly in working with generative AI models (Barke et al., 2023).

Recent studies target to enable the decomposition ability of AI models by enhancing the prompt with a series of intermediate NL reasoning steps, namely chain-of-thought (Wei et al., 2022), tree of thoughts (Yao et al., 2024), and plan-and-solve prompting (Wang et al., 2023a). However, due to the unpredictable nature of AI models, it remains challenging to determine which steps described in the prompt contribute to unexpected results.

In addition, users commonly gain proficiency in a new programming language by initially acquainting themselves with basic functions and progressively advancing towards more intricate features (Carpenter, 2021). Ordinarily, after decomposing a problem, users leverage acquired knowledge to resolve familiar sub-problems, and reserve the search for novel solutions solely for unfamiliar sub-problems. Our methodology aims to reflect this learning process on NL-to-Code translation models.

Particularly, we considered user feedback as knowledge that the translation model needs to remember following each interaction. When encountering a new NL query, the model is expected

to identify the portions of the query that have been resolved and distinguish them from the segments that necessitate code generation. The resulting composition of sub-knowledge allows for in-depth analysis of which phrases in the query lead to unexpected answers.

6.2.4 Chunking in Natural Language Processing

Shallow parsing, or chunking, involves dividing text into non-overlapping groups of syntactically or semantically related words (Abney, 1992). It is widely used in Natural Language Processing (NLP) for various types of chunks, such as named entities, noun phrases, and verbal groups (Zhang et al., 2002).

A reliable text chunker is essential for extracting relevant information from unstructured text, enabling detailed analysis in subsequent processing tasks. Different techniques, including grammar-based methods, statistical models, and ML-based approaches, have been developed for chunking tasks (Ramshaw et al., 1999; Mohapatra et al., 2021). These approaches utilize features such as part-of-speech tags or grammar templates for training.

The speedy development of Large Language Models (LLMs) has spawned a substantial amount of NLP libraries that cover a diverse array of tasks beyond chunking. Prominent libraries in this domain include NLTK[7], CoreNLP[8], scikit-learn[9], and spaCy[10]. We employed spaCy for chunking NL queries in our experiments.

6.3 One-shot Correction

This section presents a thorough explanation of our methodology, including an overview of the *One-shot Correction* workflow, and descriptions of each primary component within the workflow.

6.3.1 General Workflow

Main Components

Figure 6.2 presents the general workflow of *One-shot Correction* methodology for NL-to-Code translation models with an illustrative example. Our methodology incorporates three main components: (i) a correction data-store, (ii) an NL-to-Code generator, and (iii) a code builder.

The correction data-store collects user feedback paired with corresponding NL queries. Meanwhile, the NL-to-Code generator is a code translation model that takes NL queries as inputs and produces code snippets. The code builder is the key component designed to integrate correction information with the code generator model without requiring an additional model re-training.

[7]Natural Language Toolkit, https://www.nltk.org/.
[8]Stanford NLP Group, https://stanfordnlp.github.io/CoreNLP/.
[9]Machine Learning in Python, https://scikit-learn.org/.
[10]Industrial-strength Natural Language Processing, https://spacy.io/.
(Accessed: 19 March 2024).

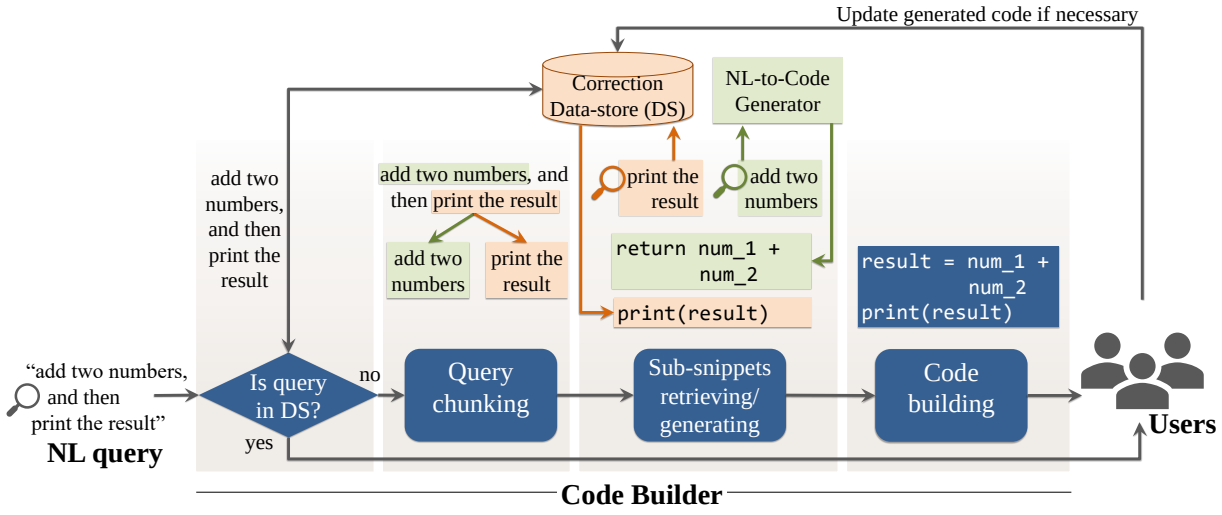


Figure 6.2: *One-shot correction* workflow for NL-to-Code translation models, exemplified with an illustrative example.

Overall Workflow

For existing queries. With each NL query, the code builder initially checks if the query already occurs in the correction data-store. If it does, the code that was previously corrected by users in past conversations is retrieved and directly returned to the users. If it is the first time users inquire about this NL query, the query undergoes several processing steps before the final code snippet is assembled.

For new queries. Initially, in the *Query chunking* step, the query is decomposed to various chunks, with each chunk representing a single task or action. Subsequently, the code builder searches for potential code snippets associated with each NL chunk by accessing the correction data-store or utilizing the NL-to-Code generator (if the chunk has no similar stored queries). We call this step *Sub-snippets retrieving/generating*.

Finally, in the *Code building* step, all the obtained code snippets are utilized to construct the final snippet before providing a reply to the user. If users make modifications to the generated code, the correction information is once again stored in the correction data-store before the next query is requested.

Illustrative example. We demonstrate here the result of each step using a typical example in Python. Assuming that the NL query is "add two numbers, and then print the result" and no prior modifications have been made by users, the query is decomposed into two chunks: "add two numbers" and "print the result".

In the subsequent step, the code builder retrieves the code snippet `return num_1 + num_2` from the NL-to-Code generator for the chunk "add two numbers" since this chunk is not present in the correction data-store. Meanwhile, the snippet for the chunk "print the result", i.e. `print(result)`, is fetched from the data-store, supposing it was corrected by users in past conversations. Ultimately, in the *Code building* step, the two code snippets are combined to generate the response, as shown in Figure 6.2.

To illustrate the applicability of our approach to different NL-to-Code translation models, we utilized existing NL-to-Code models instead of developing a new one. For simplicity, the correction data-store is structured as a dictionary, with the keys representing the embedding values of NL queries and the corresponding values indicating the corrected code. Further explanation on the NL-to-Code generator and the correction data-store employed in our experiments is presented in Section 6.4.2. The following subsections delve into a comprehensive analysis of each main phase in the code builder component.

6.3.2 Query Chunking

As mentioned in Section 6.2.4, text chunking entails grouping adjacent tokens in unstructured text into phrases based on their part-of-speech (POS) tags. In our methodology, we targeted NL queries representing pipelines of actions, where each main verb in a query indicates a task in the target code. This format resembles the DSL pipeline described in Chapter 3. Therefore, our objective in this phase is to identify non-overlapping verb-noun chunks within a query.

We used rule-based method and dependency graph to determine main verbs and construct the chunk for each verb. There are two types of main verbs considered in our methodology: (i) verbs with the POS value `VERB` (e.g. `print` the result, `calculate` the average), and (ii) auxiliary verbs (`AUX`) that are not immediately followed by other verbs (e.g. `are` inputs, `is` odd or even). Supplementary verbs do not form their own chunks (e.g. `using` Counter).

Figure 6.3 depicts a dependency graph generated by spaCy for the query "add two numbers, and then print the result". The main verbs in this query are `add` and `print`. The dependency graph reveals that all the main verbs are interconnected, while other words (e.g. `NOUN`, `ADV`) associate with their corresponding verbs. Thus, the main verb functions as the root node of its verb-phrase tree. By applying this rule to analyze the dependency graph, two verb-noun-phrases are extracted, namely "add two numbers" and "print the result". Punctuation and conjunction between main verbs are omitted in this analysis.

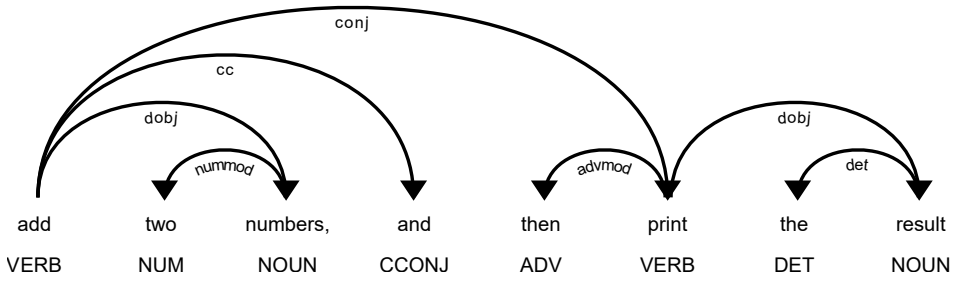


Figure 6.3: A dependency graph generated by spaCy.

It is worth highlighting the potential benefits of employing LLMs, such as GPT-3.5, in this phase. Nonetheless, our objective is to ensure the transparency of the model and the ease of comprehension for developers throughout all steps. Additionally, our evaluation results indicate that even a less complex model, when incorporated as an additional component, can already improve the efficiency of the NL-to-Code model.

6.3.3 Sub-snippets Retrieving/Generating

In our methodology, NL chunks are considered as atomic NL queries that represent a single primary task or action. The sub-snippets retrieval and generation process for an NL chunk is displayed in Figure 6.4.

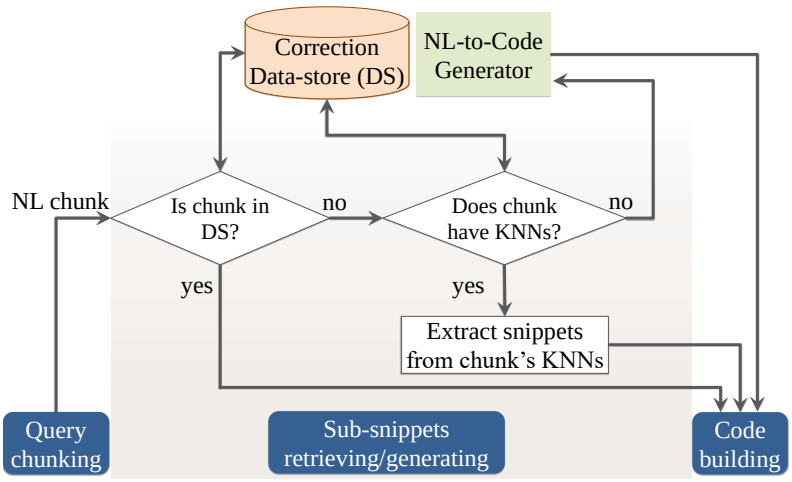


Figure 6.4: Flowchart of retrieving/generating sub-snippets for an NL chunk.

Firstly, if the NL chunk exists in the correction data-store, the related code snippets are retrieved and transferred to the *Code building* step. If the NL chunk is not present in the data-store, the k-Nearest Neighbors (KNNs) of the chunk are computed under a predetermined threshold (refer to Section 6.4.2).

Code snippets from the KNNs are extracted and forwarded to the *Code building* step. However, if there are no nearest neighbors of the NL chunk, the NL-to-Code generator is activated to generate code for the chunk and proceed to the subsequent step. Further details on code generation for NL queries or NL chunks are provided in Section 6.4.2.

Extracting Sub-snippets for an NL Chunk

In case the NL chunk has a similar NL query in the correction data-store (i.e. a nearest neighbor), the sub-snippets of the NL chunk are determined based on the sub-snippets of the phrase in the NL query that is most similar to it. Algorithm 6.1 outlines the process of extracting code snippet for an NL chunk from the corresponding code of a similar NL query.

Initially, the NL chunk is compared to the similar NL query to identify the most similar phrase, denoted as `simi_chunk` (line 2). We used the (cosine) `similarity` feature provided by spaCy to assess the correlation between the NL chunk and each chunk in the similar query, subject to a predefined threshold (specified in Section 6.4.2). Additionally, each chunk in the similar query is mapped to sub-snippets in the target code of the query, using the function named `MAP_NL_CODE` (line 3). The associated sub-snippets of `simi_chunk` are then extracted and assigned as sub-snippets for the NL chunk (line 5).

Algorithm 6.1 Extracting sub-snippet for an NL chunk from a similar query

Input: *NL chunk* (*nlc*), *similar query* (*sq*), *target code of similar query* (*sc*)
Output: *sub-snippets of NL chunk*

```
1: function EXTRACT_SUB_SNIPPETS(nlc, sq, sc)
2:     simi_chunk ← COMPARE_CHUNK_QUERY(nlc, sq)
3:     chunk_code ← MAP_NL_CODE(sq, sc)
4:     if simi_chunk in chunk_code then
5:         return chunk_code[simi_chunk]
6:     else
7:         return None
```

Mapping NL Chunks and Sub-snippets

Algorithm 6.2 represents the pseudo code for the `MAP_NL_CODE` function. We employed a rule-based approach to establish mappings between chunks in an NL query and sub-snippets in the correlative target code. Before constructing the mapping, the target code is divided into sub-snippets by analyzing its AST structure (line 3).

Algorithm 6.2 Mapping chunks in a query and its target code

Input: *NL query* (*sq*), *target code* (*sc*)
Output: *a dictionary mapping each chunk in the NL query to sub-snippets in the target code*

```
1: function MAP_NL_CODE(sq, sc)
2:     mapping ← dict()
3:     sub_snippets ← EXTRACT_SUB_SNIPPETS(sc)
4:     query_chunks ← EXTRACT_CHUNKS(sq)
5:     for sub_snippet in sub_snippets do
6:         snippet_expl ← EXPLAIN_CODE(sub_snippet)
7:         simi_chunks ← list()
8:         for chunk in query_chunks do
9:             is_simi, score ← CHECK_SIMI(chunk, snippet_expl)
10:            if is_simi then
11:                simi_chunks.append(chunk, simi_score)
12:        SORT_DESC_SCORE(simi_chunks)
13:        mapping[simi_chunks[0]] ← sub_snippet
14:    return mapping
```

We utilized tree-sitter parser[11] to obtain the AST of the target code. Sub-snippets within the target code consist of statements under the `root_node` (e.g. `import` statements) and child statements of `function_definition`. For simplicity, we required that each NL query is translated to code snippets wrapped in a `function_definition` and necessary `import` statements.

Subsequently, the NL query is decomposed into verb-noun chunks (line 4) following the method described in Section 6.3.2. To estimate the analogy between sub-snippets and verb-noun phrases, we developed a straightforward code explanation approach (line 6) that translates programming language operations and abbreviations into NL.

Afterwards, the explanation of the considered sub-snippet is compared to each verb-noun

[11] Parser generator tool, https://tree-sitter.github.io/tree-sitter/, (Accessed: 19 March 2024).

phrase, utilizing the (cosine) `similarity` function from spaCy (lines 7–11). The phrase with the highest similarity score is mapped to the current sub-snippet (line 13).

It is worth mentioning again that LLMs might be used for these NLP-related tasks. However, as we emphasized above, our goal is to investigate whether an NL-to-Code model can be enhanced by a less sophisticated method. Hence, a rule-based approach is a well-suited for this purpose.

6.3.4 Code Building

In this step, the final code is constructed by combining sub-snippets corresponding to each verb-noun phrase in the NL query. The inputs for this step include the NL query and the mapping between each phrase in the query and its respective sub-snippets. The final code encompasses sub-snippets enclosed within a `function_definition` and any required `import` statements.

This step comprises four sub-steps: (1) determining the order of sub-snippets, (2) refining sub-snippets for each verb-noun phrase, (3) renaming identifiers in all sub-snippets to ensure data-flow, i.e. naming progression from definitions to usages of identifiers in a code snippet, which is referred from semantic data-flow of Ren et al. (2020), and (4) identifying parameters for the final function. Figure 6.5 demonstrates an example of code construction from sub-snippets, using the example described in Figure 6.2.

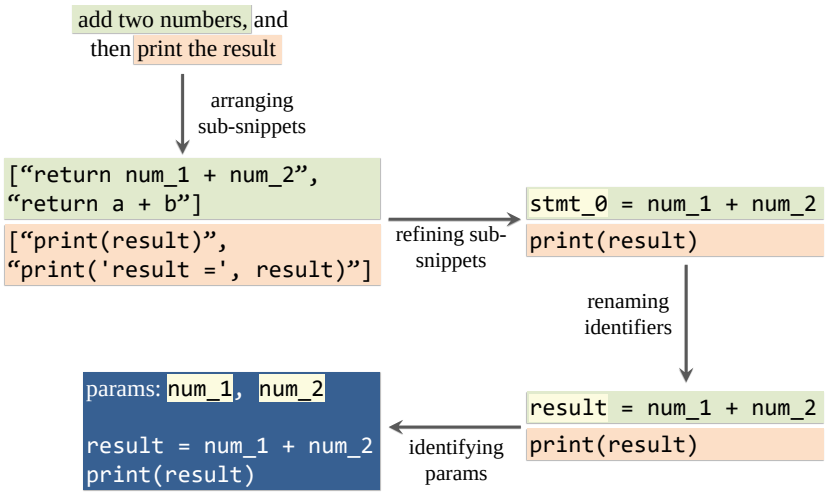


Figure 6.5: An example of building code from sub-snippets*.

* Assuming that each NL chunk retrieves *2*-Nearest Neighbors from the correction data-store, resulting in two potential sub-snippets for each chunk.

Determining Sub-snippet Order

Sub-snippets are sorted according to the verb-noun phrase order in the NL query, which corresponds to the order of related verbs in the dependency graph. The arrangement is determined by analyzing the relationship between verbs in the graph. As a result, sub-snippets associated with the root verb[12] are given priority. In Figure 6.3, the verb `add` precedes the verb `print` due to a `conj` dependency from `add` to `print`. Therefore, sub-snippets of the verb `add` (i.e.

[12]Verb with dependency label marked as `ROOT`.

`return num_1 + num_2` and `return a + b`) are placed before sub-snippets of the verb `print` (i.e. `print(result)` and `print('result = ', result)`) at the end of this sub-step (leftmost rectangles of second row in Figure 6.5).

Refining Sub-snippets

Relevant sub-snippets of each NL chunk are modified based on a set of rules. To illustrate the utility of our methodology, we initially employed three rules for sub-snippet refinement: (i) *no starting* `return` *statements*, (ii) *reducing plural statements*, and (iii) *refining between* `return` *statements*. These rules aim to minimize grammatical errors when combining sub-snippets.

(i) No starting `return` statements. This rule prioritizes non-return statements for non-last NL chunks. By default, each NL chunk is corresponded to a list of potential sub-snippets and the first item (i.e. sub-snippet(s) extracted from the *top-1* nearest neighbor) has highest priority. This is the output from the step *Sub-snippets retrieving/generating* in Section 6.3.3. The preference is maintained if the current NL chunk occupies the last position in the list of chunks achieved from the preceding sub-step (i.e. sub-snippets ordering).

Conversely, if the current NL chunk is a non-last chunk, `return` statements will have lower ranking than others. This is due to the fact that a `return` statement in the majority of programming languages will cancel other subsequent statements of the same level (e.g. same indent) within a scope (e.g. `try` statement). However, in case the non-last chunk correlates to `return` statements only, the first `return` statement is selected and converted into an `assignment`. The left operand is named using *stmt* followed by the index of the current NL chunk.

Table 6.1 displays four typical cases of refining sub-snippets for a non-last NL chunk with examples of the chunk "add two numbers". In the example depicted in Figure 6.5, "add two number" possesses the top position in the ordered list and has potential sub-snippets starting with the keyword `return` solely. Hence, the statement selected for this chunk is refined as `stmt_0 = num_1 + num_2`.

Table 6.1: Examples of refining sub-snippets for a non-last NL chunk with *2*-NNs.

Non-last NL chunk: "add two numbers", `chunk_idx = 0`		
Case	**Potential sub-snippets**	**Refined sub-snippet(s)**

No starting `return` statements	1. `sum = num_1 + num_2`
	`return sum`
	2. `result = a + b`
	`return result`
		`sum = num_1 + num_2`
		`return sum`

The `return` statement is not the first statement	1. `result = a + b` `return result` 2. `return num_1 + num_2`	`result = a + b` `return result`
The `return` statement is the first statement	1. `return num_1 + num_2` 2. `result = a + b` `return result`	`result = a + b` `return result`
There are only `return` statements in the list	1. `return num_1 + num_2` 2. `return a + b`	`stmt_0 = num_1 + num_2`

(ii) Reducing plural statements. The second rule targets to omit redundant sub-snippets of a verb-noun phrase. For simplicity, we implemented a preliminary prototype of this rule based on the direct object of the verb in an NL chunk. Nearly identical sub-snippets will be reduced if the direct object is a singular noun (i.e. spaCy `tag_` is `NN`). Contrarily, sub-snippets of the NL chunk are left unchanged if the direct object is a plural noun (i.e. spaCy `tag_` is `NNS`).

For instance, assuming that the following analogous sub-snippets are obtained for the NL chunk "get an integer input from user":

```
num_1 = int(input("Number 1:  "))
num_2 = int(input("Number 2:  "))
```

Since the direct object of the verb `get` is a singular noun (i.e. `input`[13]), only the first sub-snippet from the list of highly similar sub-snippets is preserved for building the final code, i.e. `num_1 = int(input("Number 1:  "))`.

It should be emphasized that this is an initial prototype of this rule to exhibit the concept of our approach. The reducing condition can get more complex when the plural noun is described with a specific quantity. In Figure 6.5, both of the chunks "add two numbers" and "print the result" have only one sub-snippet for each chunk, as a result from rule (i). Therefore, the rule (ii) has no effect on these sub-snippets.

(iii) Refining between `return` statements. The last rule in our primary rule set ensures that a `return` statement should be placed after other statements of the same level in the final assembled code. Namely, a non-last NL chunk should contribute non-return statement(s) to the final code. Otherwise, depending on the expression after keyword `return`, the `return` statement will be omitted or transformed to an `assignment` statement, using same technique in rule (i).

The latter case (i.e. modifying the `return` statement) happens when the part following the keyword `return` creates new values (e.g. `return a + b`, `return abs(num)`, or `return a[i]`). Alternatively, the former case arises if the after-return part is an identifier (e.g. `return sum`) or a list of identifiers (e.g. `return a, b`).

In the example exhibited in Figure 6.5, the sub-snippets of NL chunks remain unmodified after employing rule (iii). This is because there is no `return` statements left in the sub-snippets list after applying rule (i) and (ii). It is essential to mention that `return` statements nested in other code structures (e.g. `if`, `for`) are not affected by rules (i) and (iii) since the considered sub-snippets are statements right under the `root_node` of an AST and direct child statements of `function_definition`, as outlined in Section 6.3.3.

Furthermore, our primary rule set is adaptable and can be expanded for intricate cases (e.g. conditional and loop statements). We developed a configuration file to gather all the settings used in our experiments (presented in Section 6.7.1, Listing 6.4) and to conveniently select/de-select each of the refinement rules before running an experiment.

Renaming Identifiers

In this sub-step, the propagation of names within the sub-snippets is determined by analyzing code token types in the refined sub-snippets. We simplified the process by assuming that an

[13]Determined by spaCy dependency `dobj`.

identifier defined in a given statement should be utilized directly in the following statement. The inspection of sub-snippets is performed from the last sub-snippet to the first one. The underlying concept is to substitute the definitions of identifiers in the current sub-snippet with the undefined identifiers in the sub-snippet below it. Pseudo code for our algorithm is provided in Algorithm 6.3.

Algorithm 6.3 Renaming identifiers in sub-snippets

Input: *List of sorted sub-snippets* ($sub\_snippets$)
Output: *Sub-snippets with consistent names for identifiers*

```
1: function RENAME_IDENTIFIERS(sub_snippets)
2:     last_stmts ← sub_snippets[−1]
3:     last_id_defs, last_id_usgs ← GET_ID_DEFS_USGS(last_stmts)
4:     undef_ids ← last_id_usgs − last_id_defs
5:     for stmts in REVERSED(sub_snippets)[1 :] do
6:         id_defs, id_usgs ← GET_ID_DEFS_USGS(stmts)
7:         REPLACE_ID_DEFS(undef_ids, id_defs, id_usgs)
8:         UPDATE_UNDEF_IDS(undef_ids, id_defs, id_usgs)
```

The list of undefined identifiers is initialized by taking the set difference between the identifier usages and the identifier definitions in the last statement (lines 2–4). We adopted tree-sitter and our proposed method *CT3* (described in Chapter 5) to analyze token types of each token within the sub-snippets. Identifier definitions encompass variable definitions, argument definitions, and imported libraries, while identifier usages include the utilization of all the specified definitions.

Identifier definitions (id_defs) and usages (id_usgs) of each sub-snippet are determined in reversed order of sub-snippets using the same method (line 6). Subsequently, identifier definitions of the current sub-snippet are replaced by the undefined identifiers computed previously (line 7). The `REPLACE_ID_DEFS` function also considers identifier usages to handle cases where the current sub-snippet lacks identifier definitions for the one directly below it. In these cases, identifier usages of the sub-snippet are treated as identifier definitions. Finally, the list of undefined identifiers is updated to exclude the replacement (line 8)

In Figure 6.5, the list of undefined identifiers of the last statement (`print(result)`) comprises only the token `result`. Meanwhile, `stmt_0` is the only identifier definition in the preceding statement. Accordingly, after the renaming sub-step, `stmt_0` is supplanted by `result`.

Identifying Parameters for the Final Code

In the last sub-step, a list of parameters for the final function is assembled from undefined identifiers that are unsubstitutable by any identifier definitions/usages. In Figure 6.5, the tokens `num_1` and `num_2` remain as parameters of the resulting function due to the absence of appropriate identifier definitions for them within the sub-snippets.

Our methodology adheres to the principles of simplicity, interpretability, and the ability to investigate unexpected outcomes, which is not feasible with AI models. Moreover, the methodology's composability facilitates a stronger analogy between the generated code and target code with increased correction information.

Given the novelty of our approach, we aimed to illustrate the utility of the method and to highlight the main contributions. Therefore, even though our predefined rules are preliminary, they still adequately support the proposed concept. In the subsequent sections, we present our experiments and evaluation results, demonstrating how the inclusion of a relatively simple additional component can already bring benefit to a code translation model.

6.4 Experiments

In this section, we firstly reiterate our objective through two research questions. A detailed description of our experimental setup is then provided to ensure reproducibility. Finally, we present evaluation metrics used in our experiments.

6.4.1 Research Questions

We addressed the following two research questions:

RQ1: Does an interpretable, non-AI methodology enhance generative AI models? We investigated this question by proposing a rule-based methodology on NL-to-Code translation that incorporates code derived from user feedback with selectively generated code from an AI model (only as needed). Our methodology requires no explicit re-training. We conducted experiments on NL-to-Python code translation and used GPT-3.5-Turbo-0301 model developed by OpenAI as the generative AI model.

Models for comparison. To ensure a fair evaluation in the absence of existing comparable methods, we introduced an additional method that integrates correction information directly into input queries. This approach is based on the premise that GPT-3 series models tend to yield more accurate results with increased input information.

The extended input technique and our proposed methodology function similarly when the query exists in the correction data-store, as the correction information is retrieved and returned to users. However, these models differ in their response when there are similar queries in the correction data-store.

While the extended input approach simply expands the input query with information from the similar queries, our chunking methodology first decomposes the query into chunks, gathers appropriate code snippets for each chunk by examining the correction data-store or activating the NL-to-Code generator, and then constructs the final code using the collected code snippets.

In summary, our main evaluation comprises three variants: (i) **CodeGen** – code generation without correction information, (ii) **CodeGenE** – code generation with correction information integrated through extended input queries, and (iii) **CodeGenC** – code generation with correction information incorporated using our chunking methodology. The **CodeGen** model serves as the baseline. Additionally, to inspect LLM performance with our chunking strategy embedded within prompts as task descriptions, we conducted an additional experiment using GPT3.5 to directly generate code with the integrated chunking instruction, referred as **GPT35Prompt**.

RQ2: Does user feedback improve NL-to-Code models without explicit re-training? To address this question, we performed an ablation study to assess the influence

of user feedback on generated code. We compared the code generated solely by the code generator to the code produced when integrating the generator with various states of the correction data-store (defined in Section 6.4.2). This comparison allows us to determine if incorporating user feedback offers benefits to code translation models without re-training and which state of the data-store would offer the greatest advantage. Additionally, the answer of this question also resolves our final core research question (CRQ4), specified in Chapter 1, Section 1.4.

6.4.2 Experimental Setup

We performed experiments on translating NL to Python code, using available APIs and libraries as follows:

Test Cases and Scenarios

For the evaluation, we assessed the methodology using a range of NL queries, varying from basic to complex. To simplify the query chunking process, we assumed that each chunk in the query describes a single task, and the chunks are separated by comma or the term *", and then"*. While acknowledging potential artificiality in triple or more-chunk queries, our proposed structure addresses NLP ambiguity as an intermediate form between NL and Domain Specific Language. It involves an inevitable trade-off between flexibility and efficiency.

Test case collection. Due to the unavailability of a suitable test suite or benchmark tailored to our specific requirements, we developed a new test suite encompassing queries with one to three chunks along with their corresponding target code. We extracted single-chunk queries from online Python examples[14]. For multi-chunk queries, we utilized ChatGPT, a well-known model trained on an immense dataset, to form the queries.

Although ChatGPT's responses might lean toward its own biases, they remain closer to human intent and are more objective than our self-composed queries. For instance, we used the following inquiry for creating double-chunk queries, specifically related to dataframe:

> **Query:** You're a Python developer, give me examples of translating natural language queries to Python, involving dataframe. Each query contains two different tasks that are basic functions in Python and separated by comma or ", and then"

Subsequently, GitHub Copilot is employed to generate the target code for each query. GitHub Copilot is powered by Codex model[15], a descendant of GPT-3, which was trained on both NL and billions of lines of code. Hence, code generated by GitHub Copilot can serve as a reasonable reference. We thoroughly validated and modified (when necessary) each target code to ensure its validity and executability[16].

Scenario construction. For each NL query or chunk, there are five possible states of the correction data-store: (1) empty data-store, (2) identical query in the data-store, (3) non-empty

[14]Programiz, https://www.programiz.com/python-programming/examples.
[15]OpenAI Codex, https://openai.com/blog/openai-codex.
(Accessed: 19 March 2024).
[16]To ensure code generation exclusively based on the NL query, the IDE displays only one Python file.

data-store without similar queries for the inquiry, (4) similar single-chunk queries in the data-store, and (5) similar multi-chunk queries in the data-store. Accordingly, each single-chunk query involves five scenarios, while each multi-chunk query can associate up to $(1+4^n)$ scenarios, where n represents the number of chunks in the query.

The test suite should cover all the states of the correction data-store and yield sufficient results to analyze the behavior of all models, targeting to highlight the utility of the proposed method. For this reason, we gathered 47 single-chunk queries, nine double-chunk queries, and three triple-chunk queries as main inquiries, alongside 55 single-chunk queries, 20 double-chunk queries, and 14 triple-chunk queries dedicated as similar queries in the correction data-store. These queries cover 401 cases across five states of the correction data-store. Furthermore, each query chunk is guaranteed to have at least one similar single-chunk query.

Accordingly, each test scenario includes: (i) the correction data-store, (ii) NL query, (iii) target code, (iv) code generated by the NL-to-Code generator only, (v) code obtained with extended input queries, and (vi) code constructed by our chunking methodology.

Correction Data-store

For simplicity, we used a correction data-store dictionary where the keys depict tuples of embedding values of the NL query, and the values correspond to code corrected by users. Given the varying data-store states for each query, we created a data-store containing all collected NL queries and their target code, and provided a snapshot of the data-store for each test scenario.

Code Generator

We utilized a substitution of Codex model, GPT-3.5-Turbo-0301[17], for NL to Python code translation. Due to the replacement of the `CodeCompletion` feature in the Codex model with `ChatCompletion` in GPT-3.5-Turbo-0301, queries for translating NL to code are formalized as messages between system and users.

To obtain Python code from an NL query with GPT-3.5-Turbo-0301 solely (i.e. CodeGen model), we structured the messages between system and user as demonstrated in Listing 6.1.

```
1 messages=[
2   {"role": "system", "content": "You are a helpful assistant"},
3   {"role": "user", "content": "Translate the below query to Python code with the
      ↪ following constraints:
4         1. Return only Python code, no explanation, no python mark
5         2. All names in the Python code must be separated by underscores
6         Query:
7         Function <the NL query/chunk here>"},
8 ]
```

Listing 6.1: Message for translating NL query/chunk to Python code using GPT-3.5-Turbo-0301.

[17]OpenAI discontinues supporting Codex from March 23, 2023. A newer model, GPT-3.5-Turbo-0613, was released on 27 June 2023.

The response should exclude both code explanations and Python marks (e.g. `'''python`) to facilitate the extraction of code snippet. Furthermore, all variable names in the generated Python code should adhere to `snake_case` convention to enhance the mapping between NL chunks and code snippets.

As GPT-3.5-Turbo-0301 model generally gives less attention to system messages[18], to extend input queries for the CodeGenE model, we integrated the correction information to user messages, as illustrated in Listing 6.2. We refer to the documentation from OpenAI for detailed explanation of each field in the messages.

```
messages=[
  {"role": "system", "content": "You are a helpful assistant"},
  {"role": "user", "content": "Translate the below query to Python code with the following constraints:
      1. Return only Python code, no explanation, no python mark
      2. All names in the Python code must be separated by underscores
      Examples:
      <Similar queries and their target code here>
      Query:
      Function <the NL query/chunk here>"},
]
```

Listing 6.2: Message for translating NL query/chunk to Python code using GPT-3.5-Turbo-0301 with correction information combined in input queries.

Similar queries and their corrected code snippets from the correction data-store are provided as examples for the NL query and displayed in sequential order. Alternative prompting methods for user messages might impact the generated response (mentioned in Section 6.2.3). However, comparing these prompting methods is beyond the scope of this work.

Additionally, OpenAI models exhibit non-deterministic behavior, resulting in varying outputs for identical inputs. This poses certain challenges for our evaluation process, particularly when triggering the model multiple times with the same input due to the dynamic state of the correction data-store in the test scenarios. To address this issue, we adopted a dictionary-based method to accumulate and store the code generated by GPT-3.5-Turbo-0301. The dictionary uses embedding values of inquiries as keys, enabling retrieval of the corresponding generated code when an identical prompt is submitted.

Natural Language Embedding and KNNs

We employed another model from OpenAI, Text-Embedding-ADA-002[19], to embed NL queries. KNNs for each query are extracted using cosine similarity under a predefined threshold (identified in the Experiment Configuration). The accompanying function is developed by OpenAI as well.

[18] How to format inputs to ChatGPT models, https://github.com/openai/openai-cookbook/blob/main/examples/How_to_format_inputs_to_ChatGPT_models.ipynb.

[19] Embeddings with OpenAI, https://platform.openai.com/docs/guides/embeddings. (Accessed: 19 March 2024).

Experiment Configuration

Table 6.2 displays the configurations for our conducted experiments.

Table 6.2: Experiment configuration for *One-shot Correction*.

Parameter	Value
Code generator (GPT-3.5-Turbo-0301)	
`temperature`	0.9
`max_tokens`	200
`top_p`	0.9
`n`	1
`frequency_penalty`	0.5
`presence_penalty`	1.5
Correction data-store	
`knn`	2
`cosine_threshold_single`	0.15
`cosine_threshold_multi`	0.2
NL-code mapping	
`spaCy_model`	`en_core_web_md`
`query_simi_threshold`	0.5
`nl_code_simi_threshold`	0.5
`reduce_simi_snippets_threshold`	0.9

Code generator. Hyperparameters for generating Python code from NL queries using the model GPT-3.5-Turbo-0301 are outlined in the top part of the table. Specifically, a `temperature` of 0.9 and a `top_p` value of 0.9 are set to encourage the model's creativity when multiple responses are required ($n > 1$).

A `frequency_penalty` of 0.5 is assigned to penalize the frequent occurrence of repeated identifiers in code snippets, while a `presence_penalty` of 1.5 is used to prompt the model to generate a novel response each time for the same query. For simplicity, in our experiments, we considered a single response per query ($n = 1$). Further information on each hyperparameter is explained in the OpenAI documentation[20].

Correction data-store. Subsequently, queries of the correction data-store undergo KNN examination using cosine similarity thresholds of 0.15 and 0.2 for single and multi-chunk queries, respectively. Each inquiry obtains two nearest neighbors ($knn = 2$).

NL-code mapping. Ultimately, settings for obtaining sub-snippets and building the final code are specified at the bottom of Table 6.2. The spaCy model `en_core_web_md` is utilized, and another cosine similarity threshold of 0.5 is set for comparing the resemblance between chunks or a chunk and its sub-snippets. A threshold of 0.9 is employed to determine mostly identical sub-snippets for the second rule in the rule set of refining sub-snippets (Section 6.3.4).

In addition, we omitted stop words and lemmatized verbs to their base form before calculating the similarity. The thresholds in Table 6.2 are adjusted to ensure that the final code snippet is constructed successfully in a majority of test cases. As mentioned in Section 6.3.4, we gathered setting values to a configuration file to easily fine-tune all the parameters, rules, and options.

[20]Chat completion, https://platform.openai.com/docs/api-reference/chat, (Accessed: 19 March 2024).

6.4.3 Evaluation Metrics

Inspired by the study of Su et al. (2018), we assumed that users modify the generated result in the following order: (i) restructuring the code if necessary (i.e. adding, re-arranging, or removing statements), (ii) renaming identifiers and updating strings to align with the NL query. Based on this assumption, we evaluate the code obtained by different approaches using the following criteria in the descending order of priority:

1. Code validity and executability,
2. Syntax similarity between the attained snippet and the target code,
3. Data-flow correlation among the obtained results,
4. Analogy of identifier names in the code snippets.

To ensure the first criterion, we manually evaluated each test case for its correctness. The remaining criteria are assessed using CodeBLEU (Ren et al., 2020) with hyperparameters $(\alpha, \beta, \gamma, \delta)$ representing *ngram match*, *weighted ngram match*, *syntax match*, and *data-flow match*, respectively. While *ngram match* and *weighted ngram match* are targeted for the last criterion, *syntax match* depicts the syntax similarity and *data-flow match* exhibits the equivalence of data-flow.

Ren et al. (2020) recommended using the value set $(0.1, 0.1, 0.4, 0.4)$ as γ and δ have a stronger correlation with human evaluation scores. Based on the order of modifying generated code and our criteria for the evaluation, we adjusted the value set to $(0.1, 0.1, 0.5, 0.3)$. Our evaluation results show that both of the value sets follow the same trend with minimum differences. Section 6.5 presents the statistics with the amended value set (i.e. $0.1, 0.1, 0.5, 0.3$). We refer to our published data[21] for the outcomes of the other value set.

6.5 Evaluation Results

In this section, we present and analyze our evaluation results to address the two research questions from Section 6.4.1. We conclude this section by assessing the performance of an LLM with our chunking strategy outlined in the NL text prompt as task descriptions.

6.5.1 Evaluation Results by Difficulty Level

To analyze the evaluation results, we utilized the correction data-store states defined in Section 6.4.2 to determine the difficulty level for each test case and classify the results based on these levels. Each difficulty level indicates the degree of challenge in achieving the target code. The levels range from 0 to 4, representing a spectrum that includes *low*, *medium-low*, *medium*, *medium-high*, and *high* difficulty. Table 6.3 presents the definition of these levels.

[21] GitLab repository, https://gitlab.com/pvs-hd/published-code/one-shot-correction-ase. (Accessed: 19 March 2024).

Table 6.3: Definitions for difficulty levels (diff.).

Diff.	Meaning
0	The NL query is present in the correction data-store (corrt.ds)
1	Single-chunk NL query with relevant single-chunk queries in the corrt.ds
2	(i) Single-chunk NL query with associated multi-chunk queries in the corrt.ds, *or* (ii) Each chunk in the multi-chunk NL query similar to single-chunk queries in the corrt.ds
3	Each chunk in the multi-chunk NL query resembles multi-chunk queries in the corrt.ds
4	(i) Empty corrt.ds, *or* (ii) No matching queries in the corrt.ds

For instance, *difficulty level-2* involves two sub-scenarios: (i) single-chunk NL query linked to multi-chunk queries in the data-store, or (ii) a multi-chunk NL query where each chunk resembles single-chunk queries in the data-store. Meanwhile, *difficulty level-3* indicates that each chunk in the multi-chunk NL query is related to queries with multi-chunk in the data-store. Ultimately, *difficulty level-4* represents two sub-cases: (i) empty correction data-store, and (ii) no matching queries in the data-store.

Figure 6.6 presents the patterns of CodeBLEU score by difficulty level for the three models discussed in Section 6.4.1. The scores were computed for three sets: (i) all test cases (Figure 6.6(a)), (ii) correct chunking cases (Figure 6.6(b)), and (iii) incorrect chunking cases[22] (Figure 6.6(c)). The corresponding CodeBLEU scores are provided in Table 6.4.

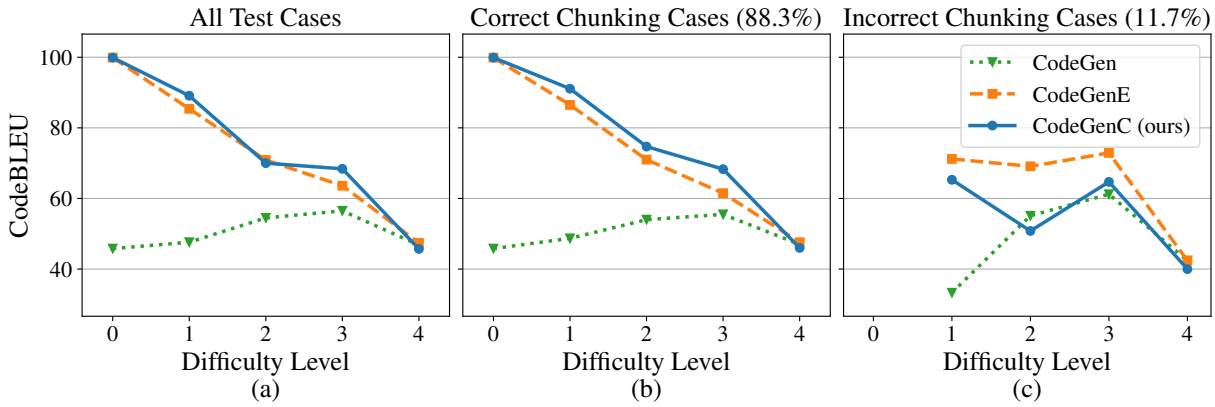


Figure 6.6: CodeBLEU scores by difficulty level on all test cases (left), on correct test cases (middle) and on incorrect test cases (right) of the chunking methodology.

Overall Evaluation

> *Finding 1.* Our *One-shot Correction* methodology generally achieves analogous or superior results compared to other approaches.

[22]There are no incorrect chunking results at *difficulty level-0*.

Table 6.4: CodeBLEU by difficulty level (diff.) across all approaches.

Diff.	All Test Cases			Corrt. Chunking Cases			Incorrt. Chunking Cases		
	CG ▼	CG'E ■	CG'C ●	CG ▼	CG'E ■	CG'C ●	CG ▼	CG'E ■	CG'C ●
0	45.8	**99.9**	**99.9**	45.8	**99.9**	**99.9**	—	—	—
1	47.6	85.4	**89.1**	48.7	86.5	**91.1**	33.3	**71.2**	65.3
2	54.5	**70.9**	70.0	54.0	71.0	**74.7**	55.1	**69.1**	50.8
3	56.5	63.6	**68.4**	55.5	61.5	**68.3**	61.2	**72.9**	64.7
4	46.8	**47.4**	45.7	47.0	**47.6**	46.0	**42.4**	**42.4**	40.0
Avg.	50.2	73.4	**74.6**	50.2	73.3	**76.0**	48.0	**63.9**	55.2

Corrt. Chunking Cases means our method yields correct results,
Incorrt. Chunking Cases indicates our method returns incorrect results,
CG = CodeGen; **CG'E** = CodeGenE; **CG'C** = CodeGenC (ours).

CodeGenC demonstrates average improvements of 1.6% and 48.6% over CodeGenE and CodeGen (i.e. the baseline model), respectively. Particularly, on test cases of *medium-high* difficulty level, CodeGenC outperforms CodeGen by 21.1%, whereas CodeGenE improves the baseline performance by 12.5% (Table 6.4, columns 2–4, diff.3).

The models CodeGenC and CodeGenE exhibit similar trends in their CodeBLEU scores with a rapid downward transition from difficulty levels 0 to 4, representing the shift from code generation with correction information to code generation without (Figure 6.6(a)). Both models significantly outperform the baseline model by a factor of 2.2 at *difficulty level-0*, where the NL query exists in the correction data-store (Table 6.4, columns 2–4). Their performance then converges to the baseline's at *difficulty level-4*.

In contrast, the standalone code generator (i.e. CodeGen) results in slight improvements from difficulty levels 0 to 3 but a decline at *difficulty level-4* (Figure 6.6(a)). Overall, the CodeGen performs worse than other models, except at *difficulty level-4*, where it slightly exceeds CodeGenC by 2.4% and lags behind CodeGenE by 1.3% (Table 6.4, columns 2–4).

To gain insights into the behavior of CodeGenE and CodeGenC models, and understand the factors contributing to performance differences, we conducted a detailed analysis primarily for difficulty levels 1 to 3 on correct and incorrect chunking cases.

Evaluation on Correct Chunking Cases

> *Finding 2.* CodeGenE may omit or become perplexed by additional information.

Difficulty level-1. Our CodeGenC model obtains accurate results on 88.3% of all test cases and consistently outperforms other models across difficulty levels 1 to 3 (Figure 6.6(b)). Particularly, in the case of *medium-low* difficulty, where the single-chunk input NL query is similar to single-chunk queries in the correction data-store, CodeGenC surpasses CodeGen by a factor of 1.9 and slightly improves upon CodeGenE by 5.3% (Table 6.4, columns 5–7, diff.1).

The latter improvement stems from CodeGenE occasionally omitting the syntax or identifier names of similar queries. The first four rows of Table 6.5 present an example for this situation.

The target code contains an `assignment` followed by a `return`, and utilizes variable names like `df` and `input_file`. While CodeGenE disregards this information, CodeGenC integrates the suggested syntax and identifier names from the similar query successfully.

Table 6.5: Examples of CodeGenE overlooks or gets confused by extra information.

Example 1: CodeGenE omits syntax and identifier names of a similar query	*NL query* : *Similar query* :	*read data from a CSV file with pandas* *retrieve data from a CSV file*

	Target code	`import pandas as pd`
		`def read_csv(input_file):`
		`  df = pd.read_csv(input_file)`
		`  return df`

	CodeGenE ■	`import pandas as pd`
		`def read_data_from_csv_with_pandas(csv_file):`
		`  return pd.read_csv(csv_file)`

	CodeGenC ●	`import pandas as pd`
		`def read_data(input_file):`
		`  df = pd.read_csv(input_file)`
		`  return df`

Example 2: CodeGenE gets confused by extra information from a similar query	*NL query* : *Similar query* :	*convert kilometers to miles* *get a number as kilometers from users, and then convert kilometers to miles*

	Target code	`def km_to_miles(km):`
		`  miles = km * 0.621371`
		`  return miles`

	CodeGenE ■	`def convert_kilometers_to_miles():`
		`  kilometers = float(input("Kilometers: "))`
		`  miles = kilometers * 0.621371`
		`  return miles`

	CodeGenC ●	`def convert_kilometers(km):`
		`  miles = km * 0.621371`
		`  return miles`

Example 3: CodeGenE encounters partial correction information for some chunks in the NL query	*NL query* : *Similar query* :	*add two numbers which are two integer inputs from users, and then print the result* *add two numbers, and then print the result*

	Target code	`def add_numbers( ):`
		`  num_1 = int(input("Enter the first number: "))`
		`  num_2 = int(input("Enter the second number: "))`
		`  result = num_1 + num_2`
		`  print(result)"`

	CodeGenE ■	`def add_two_integers(user_input_1, user_input_2):`
		`  result = int(user_input_1) + int(user_input_2)`
		`  print(result)`

	CodeGenC ●	`def add_two_numbers_print_result( ):`
		`  num_1 = int(input("Enter the first integer: "))`
		`  num_2 = int(input("Enter the second integer: "))`
		`  result = num_1 + num_2`
		`  print(result)`

Difficulty level-2 expresses cases where the single-chunk NL query is associated with multi-chunk queries in the data-store, or each chunk of the multi-chunk query resembles single-chunk queries in the data-store. At this level our model persistently excels over CodeGen by a factor of 1.4 and achieves a slight advantage over CodeGenE by 5.2% (Table 6.4, columns 5–7).

The latter increment is attributed to CodeGenE model getting confused by extra information from similar queries. Table 6.5, rows 5–8 illustrate an example of this case. While CodeGenC achieves identical syntax and variable names, code generated by CodeGenE includes a redundant statement (i.e. `kilometers = float(input("Kilometers: "))`) due to the additional chunk from the similar query (i.e. "get a number as kilometers from users").

Difficulty level-3. Notably, at the *medium-high* difficulty level, where each chunk in the multi-chunk NL query is similar to multi-chunk queries in the data-store, CodeGenC shows a 23.1% increase over CodeGen and exhibits an 11% improvement upon CodeGenE. The latter discrepancy arises due to additional information from similar queries applying to only some chunks in the input NL query.

The bottom part of Table 6.5 provides an example for this instance. The similar query "add two numbers, and then print the result" provides information that pertains to only the first and third chunks in the input query, which causes missing code lines in the snippet produced by CodeGenE. Meanwhile, CodeGenC overcomes this issue since it derives the final code from sub-snippets of each chunk in the NL query.

Evaluation on Incorrect Chunking Cases

> *Finding 3.* Our rule-based methodology underperforms compared to CodeGenE model in pure NLP tasks but facilitates in-depth analysis of inaccurate outcomes.

Difficulty level-1. Around one tenth of all test cases are classified as inaccurate chunking results. On these test cases, our model outperforms the baseline model by an average of 15%, but lags behind the CodeGenE model across difficulty levels (Figure 6.6(c)). At *difficulty level-1*, CodeGenC surpasses CodeGen by a factor of 2.0, while experiencing an 8.3% decrease compared to CodeGenE (Table 6.4, columns 8–10). This reduction results from three following factors.

Firstly, the naming convention for functions used by CodeGenC, where function names are extracted from verbs and direct nouns in the NL query, might not always align with developer preferences. In addition, the comparison between queries using cosine similarity occasionally validates similar queries with unexpected KNN order. Lastly, for simplicity, our model currently does not handle auto-detection of specific values from similar queries (e.g. two queries are similar but have different values of numbers or strings).

In contrast, CodeGenE, which is derived from a LLM GPT-3.5-Turbo-0301, acquires inherent advantages in pure NLP tasks. Additionally, CodeGenE benefits from the target code obtained through GitHub Copilot, a predecessor of GPT-3.5-Turbo-0301.

Difficulty levels 2 and 3. Analogously, the decrease of CodeGenC compared to CodeGenE at *difficulty level-2* (by 26.5%) and *level-3* (by 11.2%) is also related to NLP challenges. Tasks in these two levels include finding correct KNNs, accurately extracting the most similar chunks from

similar queries, and properly mapping NL chunks to their relevant code snippets. CodeGenC relies on rule-based approaches in performing these tasks, which faces limitations in NLP.

Difficulty level-4. Ultimately, at the *high* difficulty level, CodeGenC slightly underperforms compared to the other models with a 5.7% reduction, primarily due to the discussed naming convention for functions. It is worth noting that this intensive inspection on incorrect cases is restricted with CodeGen and CodeGenE models because of their unpredictable property.

Overall, our model, CodeGenC, demonstrates competitive performance compared to other models, despite the challenges encountered in NLP tasks. In contrast to generative AI models, our methodology offers straightforward and interpretable approaches for generating the final code, enabling thorough analysis of unexpected results and facilitating insights for potential improvements. Additionally, utilizing the explicit mapping between generated code snippets and NL chunks in a graphical user interface can simplify assessment of suggested code for users (further details in Section 6.7).

The extensive analysis of evaluation results on the entire test case dataset, spanning various difficulty levels, provides valuable information to answer the first research question introduced in Section 6.4.1.

> **A-RQ1. The rule-based methodology for NL-to-Code translation with user feedback enhances the generative AI model with comparable performance. Furthermore, the method enables thorough examination of unexpected results through its interpretable and straightforward approaches.**

6.5.2 Ablation Study

We continued analyzing the evaluation results under two aspects (i) complexity level and (ii) correct outcome ratio to examine the advantages of integrating user feedback.

Complexity Level

To study the significance of user feedback and the influence of each state of the correction data-store on generated code, we categorized the test results by complexity level. Each level describes the components required to attain the final code. These levels, ranging from 0 to 4, represent a spectrum from *low* to *high* complexity, determined by the states of the correction data-store (presented in Section 6.4.2).

For example, on test cases of *low* complexity, the NL query exists in the correction data-store, requiring only the data-store component to obtain the final code. Complexity levels 1 and 2 represent situations where the code generator is activated due to an empty correction data-store or no matching queries in the data-store, respectively. Further details for each complexity level are provided in Table 6.6.

> *Finding 4.* The correction data-store provides significant benefits when (i) the NL query is present in the data-store or (ii) each chunk in the NL query closely resembles single-chunk queries in the data-store.

Table 6.6: Definitions for complexity levels (comp.).

Comp.	Meaning
0	NL query in the correction data-store (corrt.ds), only the data-store is needed
1	Empty corrt.ds, NL-to-Code generator is activated
2	Non-empty corrt.ds but no matching queries, NL-to-Code generator is triggered
3	Some chunks in the NL query have similar single-chunk queries in the corrt.ds, NL-to-Code generator is prompted for chunks having no similar queries
4	Analogous to (3) but the similar queries are multi-chunk queries, which involves mapping NL chunk and code snippets

User feedback with CodeGenC. Figure 6.7 depicts the CodeBLEU scores of our chunking methodology (i.e. CodeGenC model) by complexity level, divided into three groups: (i) all test cases, (ii) correct chunking cases, and (iii) incorrect chunking cases.

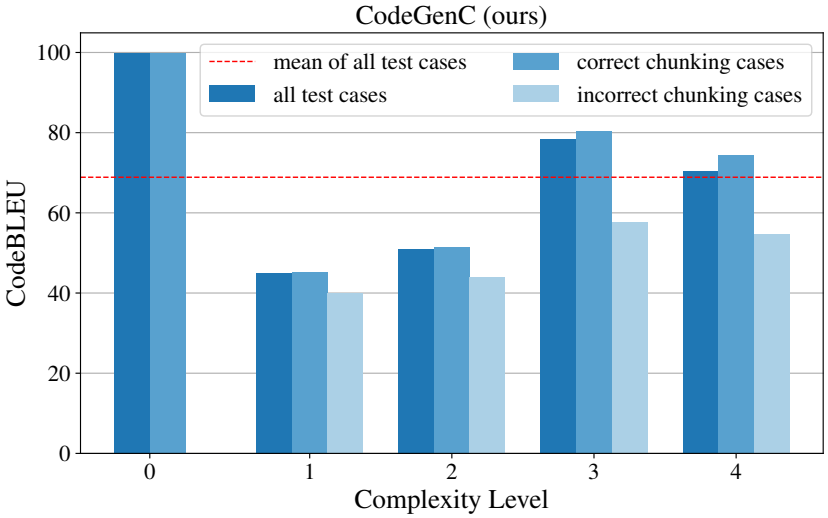


Figure 6.7: CodeBLEU scores by complexity level for our CodeGenC model.

The evaluation results demonstrate a near-perfect score of 99.9 at the *low* complexity level, indicating the presence of the NL query in the correction data-store. However, at the *medium-low* and *medium* complexity levels, the test cases occupy the lowest scores, regardless of correctness, with decrements of 34.8% and 26% compared to the average score across all test cases. This decline is attributed to the absence of user feedback for the NL query in the data-store.

Additionally, the results of *medium-high* complexity test cases slightly surpass the ones from *high* complexity by 11.4%. This can be explained by the increased complexity associated with generating the final code at the *high* level. For test cases at *medium-high* level, CodeGenC utilizes various components including the correction data-store, similarity validation between queries, and, if necessary, the NL-to-Code generator to attain the final code. Meanwhile, the *high* complexity cases require an additional component, namely the NL-Code mapping, to construct the code by identifying suitable code sub-snippets for each chunk in the query.

User feedback with CodeGenE. To assess the influence of user feedback on the CodeGenE model, we compared CodeBLEU scores of CodeGenE across complexity levels for all test cases

(Table 6.7). The outcomes align with the analysis of CodeGenC model discussed earlier. Namely, complexity levels 1 and 2 encounter the lowest scores, with decrements of 32.7% and 25.4% from the average, respectively. Test cases with *low* complexity persistently achieve almost the perfect score. In addition, results of complexity levels 3 and 4 both exceed the average with increments of 10.7% and 2%, respectively.

Table 6.7: CodeBLEU by complexity level across all test cases for CodeGenE model.

Complexity level	**CodeGenE** ■ (all test cases)
0	99.9
1	46.3
2	51.3
3	76.2
4	70.2
Avg.	68.8

Table 6.8: Correct outcome ratio and CodeBLEU for each model.

Model	**Correct outcomes**	**Code BLEU**
CodeGen ▼	**92.5%**	50.5
CodeGenE ■	90.0%	69.2
CodeGenC (ours) ●	88.3%	**69.4**

Finally, despite targeting to mimic the structure and identifier names of the corrected code in constructing the resulting code, the validity of the generated code is also an important metric. Therefore, we inspected the achieved codes by their executed output, as outlined below.

Correct Outcome Ratio

As mentioned in Section 6.5.1, for simplicity, CodeGenC composes function name based on the input NL query instead of using LLMs as CodeGen and CodeGenE. Consequently, utilizing exact match or accuracy is improper for the evaluation. Alternatively, we manually examined each obtained code snippet and validate if it yields correct output after executing. The percentage of accurate outputs over all test cases forms the *correct outcome ratio*.

> *Finding 5.* Information from user feedback can navigate the generative model to resemble the corrected code while improving some specific cases, which are previously considered as inaccurate outcomes.

Table 6.8 displays the ratios of all models alongside their CodeBLEU scores. Overall, CodeGen acquires highest percentage but the disparities between models are insignificant. Particularly, CodeGen exceeds CodeGenE and CodeGenC by 2.5% and 4.2%, respectively. In contrast, CodeGen attains the lowest CodeBLEU score, lacking behind CodeGenE and CodeGenC by 27.0% and 27.7%, respectively. In other words, the standalone AI code generator model might yield the correct output but its generated code lacks substantial alignment with user suggestions.

Specifically, Table 6.8 reveals minimal differences between CodeGenE and CodeGenC when aggregating CodeBLEU scores across all test cases. Nevertheless, at particular difficulty and

complexity levels (discussed above), CodeGenC remarkably outperforms CodeGenE, underscoring the importance of our refined evaluation in such instances.

Additionally, Figure 6.8 presents further analysis on the correct outcomes. When CodeGenC obtains accurate results from constructed code snippets, CodeGen and CodeGenE still encounter 5.4% and 9.3% of inaccurate outcomes, respectively. Notably, whereas CodeGen produces incorrect outcomes, CodeGenE and CodeGenC rectify 60% and 63.3% of these cases, turning them into correct ones, respectively.

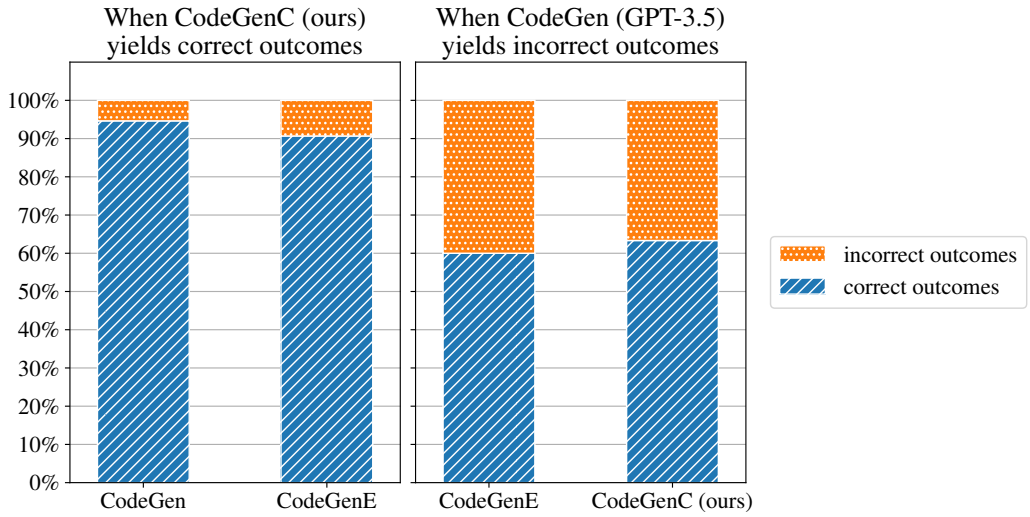


Figure 6.8: Correct outcome ratio for specific cases.

In simpler terms, generative models with extra information from user feedback can improve some cases that are original incorrect outcomes. The evaluation results unequivocally indicate the advantages of user feedback for NL-to-Code translation model, even in the absence of explicit re-training. The last two findings address our second research question outlined in Section 6.4.1.

> **A-RQ2. User feedback can bring benefits to code translation models without explicit re-training.**

Ultimately, even though prompting technique is not our main focus, it is essential to assess whether an LLM, with our chunking strategy integrated into the prompt, can perform better than our proposed model. The next subsection addresses this matter and reveals the results.

6.5.3 LLM Involvement

Primary goal reiteration. It is worth emphasizing again that besides the goal of integrating user feedback into generative AI models without extra re-training, we aimed to ensure model interpretability throughout all steps for developers (recalled from Sections 6.3.2 and 6.3.3). The latter also enables thorough analysis of incorrect outcomes. In addition, we targeted to explore the potential enhancement of an NL-to-Code model through a simplified approach. Hence, we refrained from using complex LLMs for query decomposition and chunk-to-sub-snippet mapping, and only employed them for code generation.

Furthermore, given that prompting techniques can influence the result quality in generative AI models (mentioned in Section 6.1), our approach is to assist users with standard, straightforward prompts, delegating strategy planning and reasoning to the underlying mechanism. Moreover, most of generative AI models impose constraints on prompt length or context window size (i.e. number of tokens processed simultaneously), restricting the integration of historical corrections.

LLM with chunking instruction. However, to complement our preceding evaluation, we conducted an extra experiment utilizing GPT-3.5-Turbo-0125[23] for translating NL queries to Python, incorporating our decomposition strategy as task descriptions. This experiment inspects the effectiveness of the employed models in analyzing query chunks and NL-code mapping. Consequently, we only considered scenarios with multi-chunk queries and non-empty correction data-store from the collected test cases (i.e. 39.4% of the total cases).

Listing 6.3 displays the query template utilized in this experiment. Queries from the correction data-store, serving as user-approved cases, are appended following the input NL query. The chunking strategy is outlined from lines 5 to 9. We denoted the model utilizing GPT-3.5 alongside our chunking strategy as *GPT35Prompt*.

```
1  Generate one Python function for the below NL query:
2  "Function <the NL query here>"
3  Given that users have approved the following NL queries with their target codes:
4  "<Similar queries and their target code here>"
5  To generate the Python code, do the following steps:
6      1. Divide the NL query into chunks of verb-noun phrases,
7      2. Find similar queries from the list of approved queries,
8      3. Find code snippet for each phrase in the input by either extracting it from
       ↪ the similar queries or generating it, but don't use functions from the
       ↪ similar queries,
9      4. Combine the collected code snippets to form the final code.
10 You must return only the Python code in one function, no explanation, no python mark
       ↪ .
```

Listing 6.3: Prompt template for translating NL query to Python code using GPT-3.5-Turbo-0125 and our chunking strategy as task descriptions.

Result assessment. Table 6.9 illustrates the correct outcome ratios and CodeBLEU scores of all models across test cases featuring multi-chunk queries and non-empty correction data-store. The results reveal that GPT35Prompt underperforms other models in terms of correct outcome ratio, lacking behind CodeGen, CodeGenE, and CodeGenC by 20.9%, 14.5%, and 14.6%, respectively. Besides, GPT35Prompt only surpasses CodeGen by 10.9% in terms of CodeBLEU score, while lagging behind CodeGenE and CodeGenC by 8.5% and 10.3%, respectively.

Further analysis, as depicted in Figure 6.9, confirms the inferior performance of GPT35Prompt compared to other models. Specifically, while CodeGenC (our approach) achieves correct outcomes, CodeGen, GPT35Prompt, and CodeGenE still encounter 8.6%, 29.3%, and 16.4% of incorrect outcomes, respectively. In cases where CodeGen produces incorrect results, GPT35Prompt, CodeGenE, and CodeGenC rectify 56.2%, 68.7%, and 75% of these instances.

[23] At the time of our experiment, OpenAI stopped supporting for the model GPT-3.5-Turbo-0301. GPT-3.5-Turbo-0125 is purported to exhibit greater accuracy in responding to requested formats, https://platform.openai.com/docs/models/gpt-3-5-turbo, (Accessed: 24 March 2024).

Table 6.9: Correct outcome ratio and CodeBLEU for each model over test cases with multi-chunk queries and non-empty correction data-store.

Model	Correct outcomes	CodeBLEU
CodeGen ▼	**89.9%**	57.1
GPT35Prompt ♦	69.0%	63.3
CodeGenE ■	83.5%	69.2
CodeGenC (ours) ●	88.6%	**70.6**

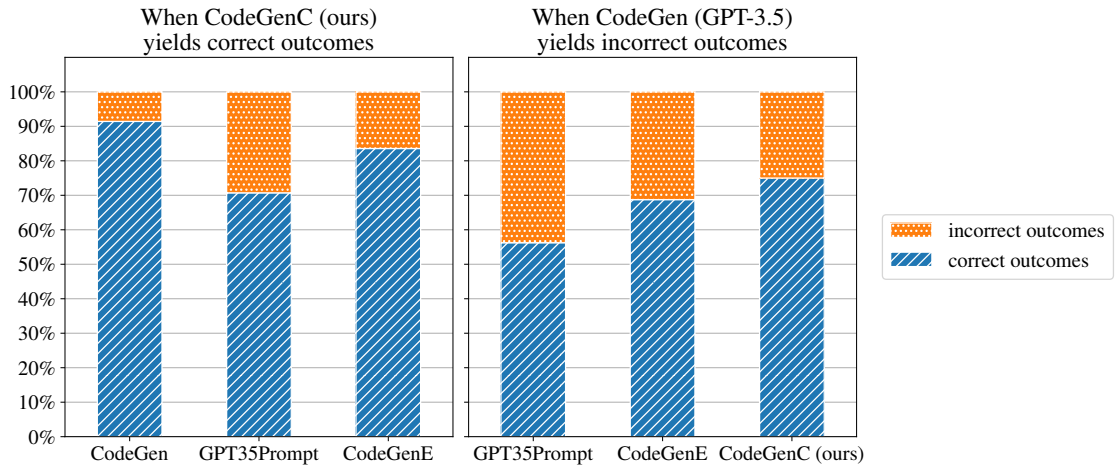


Figure 6.9: Correct outcome ratio for certain cases with GPT35Prompt taken into account.

Brief analysis. While the underlying LLMs of GPT35Prompt and CodeGenE are slightly different (GPT-3.5-Turbo-0301 versus GPT-3.5-Turbo-0125), they employ distinct prompting templates, leading to notable disparities in generating accurate final codes. This underscores the significance of prompting techniques in result quality. However, comparing prompting techniques is beyond the scope of our study.

We briefly examined the failed cases of GPT35Prompt and discovered that GPT35Prompt also experiences the similar shortcomings as CodeGenE (e.g. overlooking or becoming confused by additional information, as shown in Table 6.5). Additionally, 55.1% of the incorrect outcomes stem from GPT35Prompt generating code that uses functions defined in queries from the correction data-store, without including these function definitions into the final code. Even after adjusting the prompt template in Listing 6.3 to explicitly address this issue[24], the incorrect outcomes persist.

It is worth noting here that we considered NL-to-Code generation individually for each query. The corrected codes refer to preceding corrections but are not available to users at the moment of executing the prompts. An enhancement for this matter is discussed in Section 6.6.2. For simplicity, we excluded the analysis of GPT35Prompt results based on individual difficulty and complexity levels.

Ultimately, we anticipate that advanced prompting techniques, such as chain-of-thought (Wei et al., 2022) and tree of thoughts (Yao et al., 2024), could improve the LLM outcomes. Nonetheless, despite detailed strategy descriptions, the inherent *black-box* nature of LLMs still hinders

[24] By extending step 3 in the description with the guidance "`If you use functions defined in the similar queries, you must add these function definitions into the final code`".

thorough analysis of unexpected results, making it challenging to pinpoint which step in the strategy description causes the failed cases.

6.6 Discussion

In this section, we discuss threats to validity of our experiments, as well as challenges and potential enhancements for our methodology. We conclude the section with our response to the final core research question, specified in Chapter 1 , Section 1.4.

6.6.1 Threats to Validity

We analyzed threats to validity of our work as follows:

Test suite. A custom test suite was developed for the experiments due to the absence of a suitable existing test suite. Though our dataset is not as extensive as those for AI model training, it sufficiently demonstrates our methodology's utility. However, the inclusion of an official benchmark would enhance the effectiveness of the proposed approach.

In future work, we intend to incorporate more complex test cases, probably by refining Q&As from programming forums. Furthermore, the lack of probability logs from Codex model in the response of ChatCompletion feature (GPT-3.5-Turbo-0301) raises questions about the likelihood of the code returned in the first response being the most probable one.

Language specificity. The algorithm for mapping NL chunks and code snippets in the *Code building* step is currently implemented exclusively for Python. However, the identification of code token types is based on AST analysis and token relationships, which vary slightly across programming languages.

Besides, the algorithm focuses on critical token types shared among programming languages, such as variable definition and usage. Determining these token types in other languages (e.g. Java) is even less complicated than for Python due to Python's dynamic typing. Therefore, we anticipate that our results will be applicable to other programming languages. Additionally, it is worth mentioning that the parser used in the *Query chunking* step is specifically for English language. Nonetheless, multilingual NLP is outside the scope of this work.

Model comparison. Our experiments employed GPT-3.5-Turbo-0301 model, which has demonstrated significant advancements in NLP tasks. However, being a beta version and subject to frequent updates, minor adjustments may be necessary to accommodate changes in its APIs. Furthermore, due to the lack of directly comparable models, we compared our methodology with extended input queries on GPT-3.5-Turbo-0301. We assume that comparing other approaches that utilize the chunking method would further validate the concept of our methodology.

Evaluation metrics. Besides manually examining the validity of generated code, we adopted CodeBLEU as an evaluation metric due to its popularity in code generation models. Although ChrF has been proposed as an alternative (Evtikhiev et al., 2023), it does not fully consider the specifics of working with source code. As our experiments prioritize the syntax of the generated code (as discussed in Section 6.4.3), CodeBLEU with the mentioned settings remains suitable for our purposes.

6.6.2 Challenges and Potential Enhancements

Given the novelty of our proposed methodology, we outline below challenges while developing the approach, alongside potential improvements which can make our concept applicable to more intricate use cases.

Scalability Support

Multi-users and large datasets. To illustrate the utility of our methodology, we collected user feedback in a dictionary with embedding values of the input queries as keys and the corrected code snippets as corresponding values. Subsequently, similar queries of each input are retrieved using KNN technique, by comparing similarity between the input and all existing queries in the data-store. This simple setup serves its purpose in exhibiting the advantages of integrating user feedback into generative AI models without re-training. However, adapting this method to multi-user systems and large datasets necessitates upgrading the correction data-store structure.

Particularly, users usually refer to their own naming patterns (while aligning to coding convention) for identifiers, requiring the separation of correction information stored for individual users or only shared within user groups. Furthermore, a function generated from an input query can be adopted multiple times at different locations within a program, each with distinct sets of variable names. Consequently, various versions of function customization should be stored instead of employing a single record for each query and overriding previous corrections.

Dynamic Sparse Distributed Memory. The presence of numerous users can result in data expansion, necessitating scalability features in the correction data-store architecture. To address this, a potential solution is employing Dynamic Sparse Distributed Memory (DSDM) introduced by Pourcel et al. (2022), an extension of Sparse Distributed Memory (Kanerva, 1992).

DSDM begins with an empty memory space and incrementally adds new address nodes based on input patterns, dynamic write radius, and current memory space state. Query content is retrieved from specific memory nodes using a `softmin` function that considers the distance between the query and other query addresses. Integrating DSDM into the *One-shot Correction* approach may enhance the correction data-store's capacity, mitigating scalability challenges.

Flexible Rule Selection for Code Building

Even though we deployed a configuration file (outlined later in Listing 6.4) to centrally manage rules for refining sub-snippets, the inclusion of rules for renaming identifiers, determining parameters for the final code, and handling multi-input queries would be beneficial. Moreover, a flexible selection mechanism for these rules should be employed based on the input query and corrected codes from similar queries.

Identifier renaming. For instance, when renaming identifiers within combined sub-snippets by prioritizing the last statement (Section 6.3.4), situations arise where the final code snippet lacks desired names, compared to the corrected code. This occurs because desired names initially appear atop the statement list but are subsequently replaced by identifier names in statements below. Consequently, a flexible activation of renaming rules (top-down or bottom-up) should be

determined based on the positions of chunks in the input query receiving similar queries from the correction data-store.

Furthermore, to exemplify the proposed chunking concept, we simplified the renaming process by assuming that identifiers defined in one statement are directly utilized in the subsequent statement. A potential enhancement to diminish this assumption involves (i) preserving the data-flow of each variable in every code snippet, (ii) analyzing the purpose of each variable definition and usage, and (iii) bridging the data-flow gap between code snippets. These steps may require NL chunks, their associated code snippets, and the input query as inputs, suggesting the consideration of a more intricate rule or approach.

Parameter determination. Additionally, as we targeted generating final codes comprising code snippets enclosed within a function definition with requisite import statements, the current parameter identification rule for the final function suffices to illustrate the method's concept. However, in case the input query requests multiple functions or omits this requirement, the rule should be adjusted accordingly, which is technically feasible by identifying the scope of variables besides their definitions and usages.

Multiple input queries. Ultimately, our proposed approach addresses NL-to-Code cases individually, as depicted with a GUI in Section 6.7. However, when applying this method to a code file containing existing NL queries and their relevant code snippets, or when dealing with inputs featuring multiple NL queries, consideration should be given to previously generated code when constructing the outcomes.

In such instances, a rule should prioritize suggested code snippets using functions defined from prior queries over code snippets that redefine these functions. Preceding queries and their codes can be directly injected into the input query, forming a multi-turn programming pipeline, similarly to the study described by Nijkamp et al. (2022).

6.6.3 Response to CRQ4

As delineated in Chapter 1, Section 1.4, our final core research question (CRQ4) delves into leveraging user feedback to refine NL to code models without explicit re-training, a challenge posed by the inherent characteristics of generative AI models. The comprehensive exploration detailed in Section 6.5, accompanied by five corresponding findings, together with the discussion above furnish sufficient evidence to address the last core research question as follows:

> **A-CRQ4. User feedback not only enhances NL to code models without extra re-training, but also enables auditing of code provenance, facilitating in-depth analysis of unexpected model outcomes.**

6.7 One-shot Correction GUI

In this section, we introduce our preliminary GUI[25] built on the *One-shot Correction* method. The GUI exhibits the practicality of our proposed concept in simplifying code customization and assessment for users. Main features of the GUI are demonstrated at the end of this section.

6.7.1 An Overview of the One-shot Correction GUI

We drew inspiration from the work of Su et al. (2018) on building an application with fine-grained user interaction for code modification. With each code token in a returned code, we determined its token type and a list of alternative values, which are extracted from other suggested codes for the same token type. Figure 6.10 presents the general scenario of using the GUI.

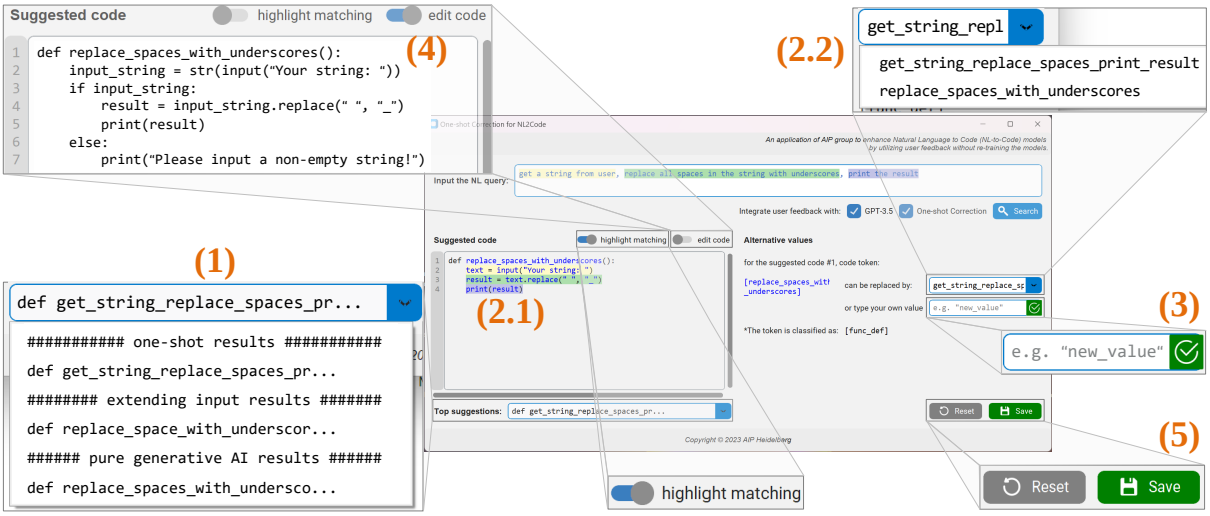


Figure 6.10: General scenario of using the *One-shot Correction* GUI.

General scenario. After initiating a search with an input NL query, users can perform the following actions: (1) choose displayed code from a list of returned code snippets, (2.1) select a code token under *Suggested code* by clicking on it and (2.2) change its value from the list of substitute values, (3) type a new value for the code token if the preferred value is not on the list in step (2.2), (4) directly modify the code if restructuring is necessary, and (5) save the modification for subsequent inquiries.

Model selection. By default, user modification is integrated with both options, *GPT-3.5* and *One-shot Correction*, which are corresponding to the CodeGenE and CodeGenC models mentioned in previous sections. Deselecting these options results in the code snippet using solely the CodeGen model (i.e. without user feedback). Besides, for each code token, we also provided its token type as an extra information for users.

Utility features. Notably, the *highlight matching* option associates input query chunks with sub-snippet(s) of the displayed code in the *One-shot Correction* case. For other cases (i.e. standalone code generator and extending input), the whole input query and its code are marked

[25]The GUI was developed using CustomTkinter, https://customtkinter.tomschimansky.com/, (Accessed: 19 March 2024).

without separation (illustrated later in Section 6.7.2). We expect that this explicit mapping can facilitate users in comprehending and validating the generated code.

Additionally, by modifying the configuration file (partially presented in Listing 6.4), users can manipulate the state of the correction data-store (line 8), filter important code token types (lines 9–11), and adjust hyperparameters used in each model (lines 2–5). We published these setting values together with our source code[26].

```
1  ...
2  "embedding": {...},
3  "nl2code": {...},
4  "correction_ds": {...},
5  "nl2code_with_correction": {...},
6  "gui": {
7    ...
8    "corrt_ds": ["all"],
9    "token_types": {
10     "syntax_types":["arg_def", "attribute", "class_def", "class_usg", "const_float", ↩
       ↪  "const_int", "const_str", "exception", "func_def", "func_usg", "imp_alias",  ↩
       ↪ "imp_lib", "imp_sublib", "keyword", "method_def", "method_usg", "var_def", " ↩
       ↪ var_usg", "znknown"],
11     "filter": [true, true, true, true, true, true, true, true, true, true, true,  ↩
       ↪ true, true, true, true, true, true, true, true]},
12   ...
13 }
14 ...
```

Listing 6.4: Relevant fields of the configuration file on handling the state of the correction data-store and refining code token types.

For instance, possible values for `corrt_ds` involve `"all"` (all gathered queries), `"all_x"` (a collection of all x-chunk queries, $x \in [1, 2, 3]$), `"all_x_excl"` (all x-chunk queries excluding the current target query), and `"task_x_y"` (the x-chunk query with index y). An example of code generation with two different states of the correction data-store is demonstrated in the subsection below. Furthermore, to prefer specific token types, users can simply enable or disable the corresponding flag of the token type (Listing 6.4, line 11). These types are determined based on our *CT3* schema, discussed in Chapter 5.

6.7.2 A Demo of Main Features

We introduces below some test cases adapting the *One-shot Correction* GUI for (i) generating code snippets, (ii) customizing the achieved code by modifying identifier values or code structure, (iii) validating the obtained code with NL-code mapping, and (iv) manipulating correction data-store for exploring the underlying generation models.

Code Generation

By clicking the *Search* button, code snippet(s) of the corresponding input NL query will be retrieved/produced. There are two options to combine user correction with the final results,

[26] GitLab repository, https://gitlab.com/pvs-hd/published-code/one-shot-correction-ase, (Accessed: 19 March 2024).

namely integrating with (i) *GPT-3.5* and (ii) *One-shot Correction*. These options are equivalent to the aforementioned CodeGenE and CodeGenC models. When both of the options are unselected, the original CodeGen model will be activated.

Assuming that the input query is "get a string from user, replace all spaces in the string with underscores, print the result". Table 6.10 presents the first nearest neighbor (*1*-NN) retrieved from the correction data-store for each chunk of the input query. The resulting code snippet after triggering the search feature with both selected options *GPT-3.5* and *One-shot Correction* is displayed in Figure 6.11.

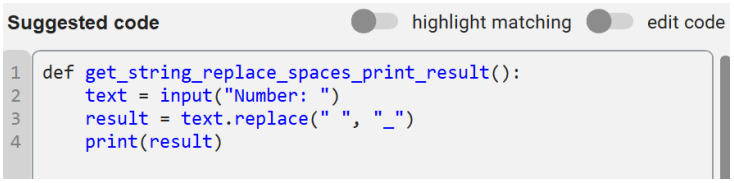
Table 6.10: *1*-NN of the input query at the beginning.

Chunk	***1*-NN from the correction data-store**	**Code snippet of the *1*-NN**

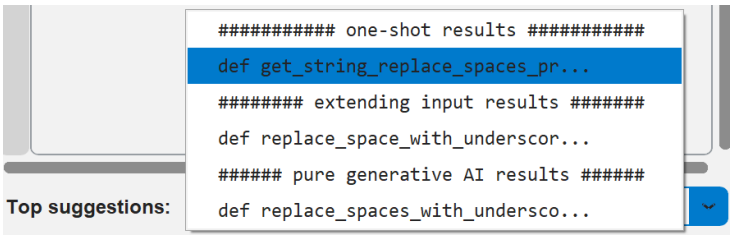
get a string from user	get an input from users	`def get_input():`
		`  input_user = input("Number:  ")`
		`  return input_user`

replace all spaces in the string with underscores	replace all spaces in a string with underscores	`def replace_spaces(text):`
		`  replaced = text.replace(" ", "_")`
		`  return replaced`

print the result	print the result	`def print_result(result):` `  print(result)`



(a) Top-1 suggested code.



(b) List of recommended code snippets.

Figure 6.11: Searching the example input query for the first time.

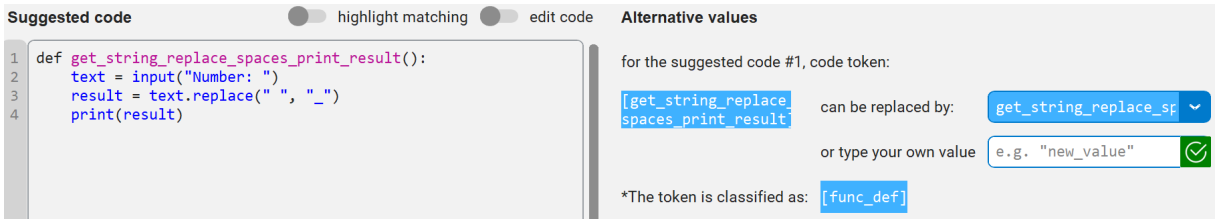
Specifically, the top-1 code snippet (in this case, code generated with the *One-shot Correction* option) is shown in Figure 6.11(a) while the list of attained snippets (grouped by models) is exhibited in Figure 6.11(b). Notably, since handling specific numbers or strings is beyond the scope of our work (mentioned in Section 6.5.1), the string utilized in line 2 of the suggested code (i.e. `"Number:  "`) needs to be refined to suit the input query.

Code Customization

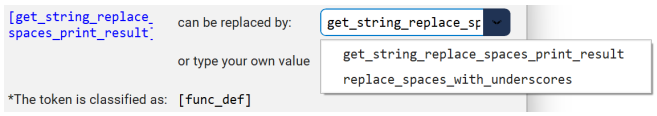
To update the code snippet for the input query, users can perform the following steps:

1. **Changing suggested code**. By browsing items in the list of recommended snippets (i.e. *Top suggestions*, Figure 6.11(b)), users can inspect code generated by each model and choose suitable code structure alongside identifier names.

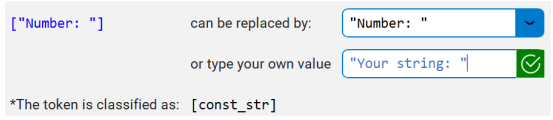
2. **Choosing alternative values**. Subsequently, users can customize the displayed code by selecting substitute values of each code token. Figure 6.12(a) presents the screen capture of the GUI right after clicking on the function code token. Relevant fields of the code token (i.e. current value, alternative values, and token type) are highlighted briefly to capture user attention. Consequently, users can choose another value for the function name from the list (Figure 6.12(b)).



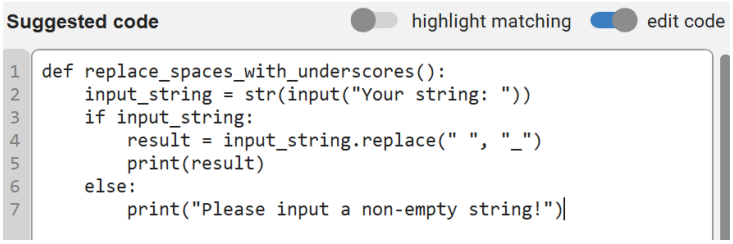
(a) Clicking on a code token to retrieve relevant information.



(b) List of alternative values.



(c) Inputting a new token value.



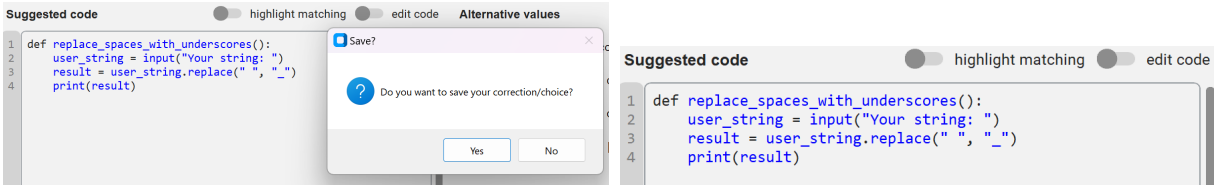
(d) Directly modifying the code snippet.

Figure 6.12: Recommended steps to customize the displayed code snippet.

3. **Inputting a new token value**. In case there is no preferred value in the replacement list, users can type a new value for the code token and click the check button next to the text input field or simply enter to apply the new value (Figure 6.12(c)).

4. **Modifying directly**. Ultimately, users can also write their own code if the current code structure does not fit their requirements or expectation (Figure 6.12(d)).

5. **Saving the correction**. In the end, users should save their feedback for future reference. For instance, given that a user updated the name of the function displayed in Figure 6.12(a) to `replace_spaces_with_underscores`, modified the string from `"Number: "` to `"Your string: "`, and changed the variable `text` to `user_string`, Figure 6.13 reveals the suggested code on the second time of submitting the same input query.

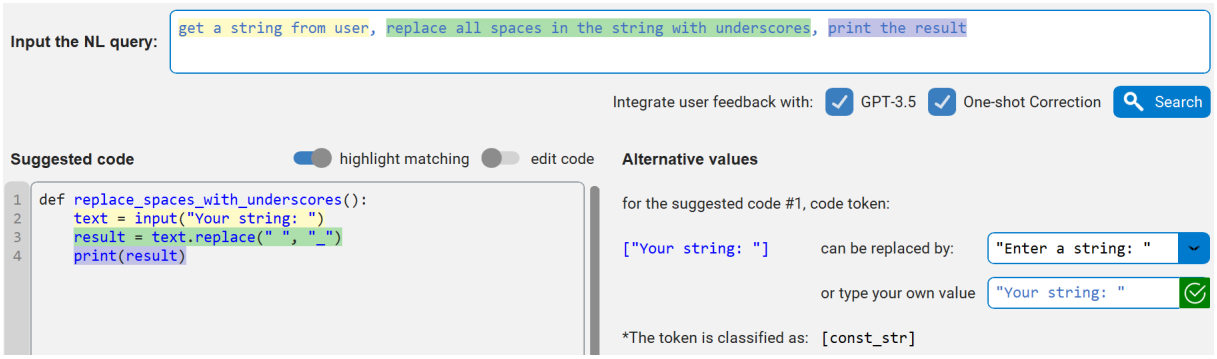


(a) Saving the correction information. (b) Inquiring the same input query after saving.

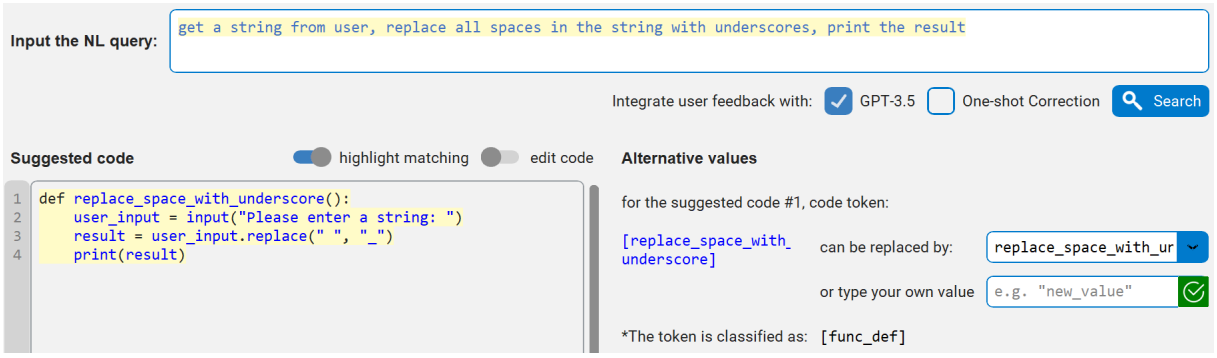
Figure 6.13: Saving the correction and inquiring the same query again.

NL-code Mapping

We developed the feature of highlighting NL-code mapping to illustrate the utility of our proposed approach in simplifying code assessment for users. Particularly, by activating the switch *highlight matching* of the GUI, users can inspect the correlation between each chunk in the input query and its code snippets constructed by the *One-shot Correction* method (Figure 6.14(a)). This feature does not work for results obtained using the two remaining methods due to their lack of decomposition information (Figure 6.14(b)).



(a) Code built and edited with the *One-shot Correction* technique.



(b) Code generated by extending input query with user feedback.

Figure 6.14: Activating the highlight matching feature.

Manipulating the Correction Data-store for Testing

Queries in the correction data-store can be manually managed through the configuration file (represented in Listing 6.4) by specifying the query indices (details in Section 6.7.1). This feature is utilized when users prefer to find the suggested code of the same input query with various states of the correction data-store. For example, given the above input query, assuming

that the correction data-store now contains only one query, “get a string input from user”, as depicted in Table 6.11.

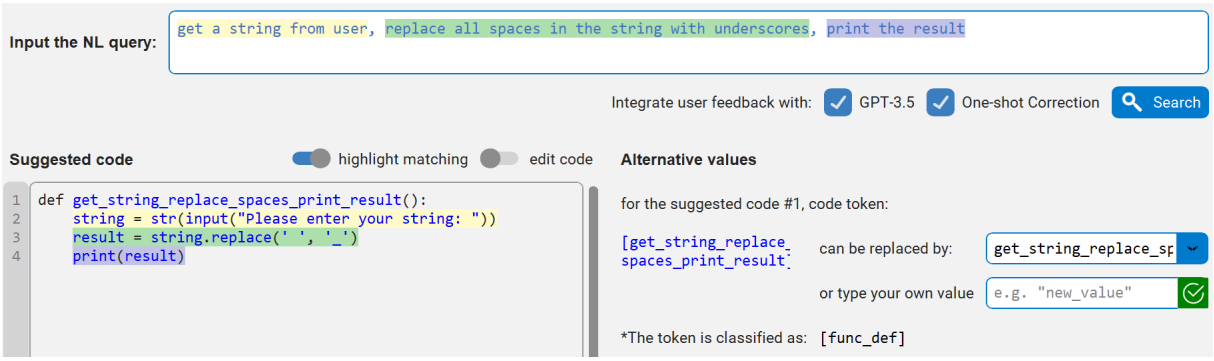
Table 6.11: *1*-NN of the input query after updating the correction data-store.

Chunk	***1*-NN from the correction data-store**	**Code snippet of the *1*-NN**

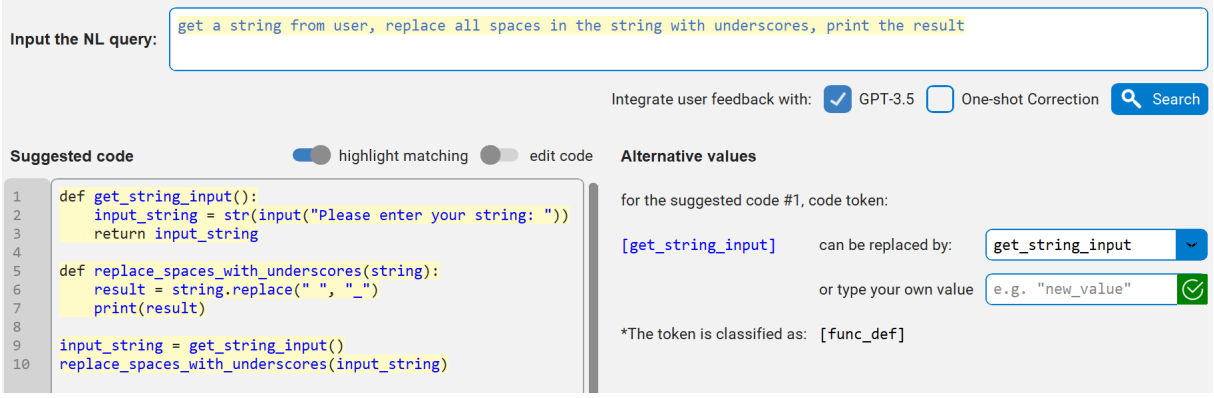
get a string from user	get a string input from user	`def get_string_input():`
		`  input_string = str(input("Please enter your string:  "))`
		`  return input_string`

replace all spaces in the string with underscores	None	None
print the result	None	None

While there is no preceding modification for the input query, the code snippet constructed by the *One-shot Correction* method (i.e. CodeGenC model) is slightly updated as illustrated in Figure 6.15(a). Notably, the structure of the code snippet generated by extending the input query (i.e. CodeGenE model) is significantly changed to two separated functions followed by two code lines, as shown in Figure 6.15(b) .



(a) Code constructed with the *One-shot Correction* technique.



(b) Code generated by extending input query with user feedback.

Figure 6.15: Generating code snippet for the same input query after manipulating queries in the correction data-store.

We utilized the same technique to produce diverse states of the correction data-store for each input query in the experimental evaluation, discussed in Section 6.4.2.

6.8 Summary

In this dissertation's final contribution, we proposed a methodology named *One-shot Correction* to incorporate user feedback into generative AI models without re-training. Evaluation results illustrate competitive performance compared to other models, despite challenges in NLP tasks. Our methodology enables thorough examination of unexpected results through straightforward approaches and facilitates insights for potential improvements.

Besides, we demonstrated that user feedback significantly enhances code translation models without re-training, addressing the last core research question (CRQ4) defined in Section 1.4 of Chapter 1. We published the test suite used in our experiments, evaluation results, and source code of the methodology[27].

Furthermore, a preliminary GUI with fine-grained user interaction in code customization was also implemented to sketch the utility of our proposed approach in practice. Further work encompasses extending the method to other programming languages and large datasets, which includes upgrading the correction data-store structure for scalability (e.g. using Dynamic Sparse Distributed Memory). Moreover, exploring flexible rule selection at each step in the methodology for complex inquiries is a promising direction.

Ultimately, this chapter concludes our array of methodologies for tackling the *programming barrier* </> and contributes to enriching libraries for code generation tasks via published source code and experimental data. The latter mitigates the *reuse problem* ♻ specified in Chapter 1, Section 1.1. In the subsequent chapters, we wrap up the dissertation and outline potential future research directions aligned with our work.

[27] GitLab repository, https://gitlab.com/pvs-hd/published-code/one-shot-correction-ase, (Accessed: 21 March 2024).

Part IV

Conclusions

Summary

Chapter 7

Driven by the rapid advancement of data science, characterized by the proliferation of diverse tool-kits, platforms, and programming languages, we target to accelerate scripting tasks for both domain experts and developers. This chapter reiterates our research objectives and strategy, recalling our contributions throughout the dissertation.

Research objectives. Our purpose is to resolve three main problems, namely *programming barrier* </> , *reuse problem* and *scalability problem* . The *programming barrier* </> occurs when domain experts have to learn a myriad of data analysis tools before using them. The *reuse problem* arises when users adjust implementations of well-known algorithms and techniques across platforms and programming languages. Finally, the *scalability problem* emerges when users transition between small and large datasets, which necessitates the deployment of complex data structures, new libraries, and potentially re-implementation.

Research strategy. In particular, the *programming barrier* </> is tackled through harnessing the benefits of Domain-Specific Languages (DSLs) and advanced methods of code completion. Specifically, *low-code* techniques and ML-based approaches are employed to ease this problem. Meanwhile, the *reuse problem* is alleviated by facilitating creation and utilization of multiple libraries containing domain-specific operations. Lastly, the *scalability problem* is handled by unifying APIs for sequential and massively-parallel processing. Additionally, certain functions in the libraries mentioned in the *reuse problem* are also implemented in a scalable fashion.

Contributions. Aligned with our objectives and strategy, our work emphasizes both practical applications and research contributions. The practical applications include releasing a VSCode extension called *NLDSL*, along with prototypical tools, and libraries. Our research contributions comprise a code recommendation architecture named *Extended Network*, a refined evaluation methodology known as *Code Token Type Taxonomy (CT3)*, and a user feedback-driven code generation approach referred to as *One-shot Correction*. These contributions were designated to answer the following core research questions, respectively:

- **CRQ1**. Do embedded *external* DSLs offer benefits for implementing data analysis tasks?
- **CRQ2**. Do ensembles of ML-based recommenders improve the accuracy for code completion approaches?
- **CRQ3**. Do traditional aggregated evaluation methods reveal useful information for comparing and characterizing code completion approaches?
- **CRQ4**. Can user feedback enhance NL to code models without explicit re-training?

NLDSL Extension

While DSLs exhibit their utility in programming facilitation, developing such languages presents challenges to developers and end-users due to the need for expertise in both programming and domain knowledge. Consequently, the matter in **CRQ1** shifts to streamlining DSL development for end-users and developers. Chapter 3 introduces *NLDSL*, our VSCode extension designed for this purpose. The *NLDSL* extension is freely accessible on the VSCode Marketplace, with three versions for Windows, macOS, and Linux.

Core functionalities of the extension involve DSL customization and development via various means. For instance, a DSL development wizard enables users to create new DSLs from Excel or `tx` templates and to manage DSLs directly within VSCode. Customized DSLs can be easily shared by distributing modified templates. The extension also includes predefined DSLs tailored for Pandas, PySpark, TensorFlow, and PyTorch.

In addition, the *NLDSL* extension provides code completion support for DSL operations (i.e. suggesting next DSL tokens) and statements (i.e. translating DSLs to programming languages). Notably, advanced features, such as type provider, path completion, in-editor documentation, and library initialization, enhance the usability for code completion tasks.

Our preliminary evaluation on the extension's features highlights their capability in assisting developers and end-users to define DSLs using average computing resources. Moreover, statistics from disseminating the extension reveal positive response from the community (more than $17k$ installations and counting).

The *NLDSL* extension mitigates the *programming barrier* </> by leveraging the advantages of DSLs. Meanwhile, DSL development features, specifically the developed wizard, contribute to address the *reuse problem* . Furthermore, the *scalability problem* is lessened by aligning DSL grammars for both sequential and parallel data processing (e.g. Pandas and PySpark), and for ML operations across platforms (e.g. TensorFlow and PyTorch).

Extended Network

Our first research contribution focuses primarily on dealing with the *programming barrier* </> through code recommendation models. To tackle **CRQ2**, we proposed an ensemble approach for predicting next code tokens in dynamically typed languages, elaborated in Chapter 4. For illustrating the approach, we introduced a model named *Extended Network model*, adopting classical and neural ML models, namely Probabilistic Higher Order Grammar (PHOG) and Pointer Mixture Network, respectively.

This combination stems from the effectiveness of the Pointer Mixture Network in predicting subsequent code tokens for dynamically typed languages, while reducing Out-of-Vocabulary (OOV) suggestions. However, the Pointer Mixture Network is limited in its ability to predict OOV words beyond the current context. Meanwhile, PHOG operates as a probabilistic model for code recommendation, utilizing production rules from a context-sensitive grammar with minimal constraints on vocabulary size and long-range dependencies. We employed PHOG to handle the cases where the Pointer Mixture Network fails to predict OOV words.

Our evaluation results show the enhanced accuracy of the *Extended Network model* compared to its individual components. Further performance improvements are achievable by augmenting the neural network with an additional layer. Here, the term "*Extended Network*" refers to an adaptable ensemble-like architecture capable of integrating various models and future methodologies as ensemble components.

We also conducted an ablation study on the *Extended Network model* to assess the effect of its components, which are RNN and Pointer Network (from Pointer Mixture Network), and PHOG. This study reveals the dominance of the RNN component compared to others. However, the performance of these components on specific prediction cases like keywords, parameters, or function names remains unclear, prompting our second research contribution.

Code Token Type Taxonomy

Given the diverse demands of code completion across token types, influenced by preferences of developers, evaluating the impact of each code completion model (e.g. within ensemble-like architectures) on specific cases would provide insights for further refinement. Our **CRQ3** and Chapter 5 are dedicated to this concern. Initially, we compared the state-of-the-art of ML-based code completion approaches, highlighting the prevalence of conventional aggregated evaluation methods among these models.

Subsequently, we proposed a refined evaluation methodology known as *Code Token Type Taxonomy (CT3)*. It first identifies multiple dimensions for code prediction (e.g. syntax type, origin, length), categorizing code tokens into meaningful types along each dimension. Evaluation metrics are then computed for each type, facilitating in-depth examination on performance of code completion models. To illustrate the utility of this approach, we compared the code completion accuracy of a Transformer-based model in two variations: with closed and open vocabulary, using both aggregated and refined evaluations.

Our empirical study indicates that *CT3* effectively characterizes and compares the accuracy of different approaches, in contrast to traditional aggregated evaluation. Furthermore, results from the refined evaluation reveal that an open vocabulary notably increases the accuracy of the Transformer-based model in code completion tasks, particularly in handling usage of defined variables and literals. Besides, our review of state-of-the-art ML-based models underscores the need for standardized benchmarks in the code completion domain.

Regarding the aforementioned three main issues, *CT3* offers valuable insights for improving code completion models through extensive result analysis, directly targeting the *programming barrier* </>. Moreover, the *reuse problem* is alleviated by our published source code and data, enabling reproducibility and experimentation with alternative models. Finally, the *scalability problem* is partially resolved through the parallel implementation of *CT3* functions.

One-shot Correction

In our final research contribution, we shifted our focus from suggesting next code tokens to simplifying programming with Natural Language Processing (NLP) techniques, particularly in

NL to code models. Motivated by the omission of user feedback in widely adopted generative AI models, we introduced a methodology for incorporating user feedback into these models without re-training, addressing the last core research question, **CRQ4**. We referred to our approach as *One-shot Correction* and detailed it in Chapter 6.

Essentially, we employed decomposition techniques to segment code translation problem into sub-problems. The final code is assembled using code snippets from each query chunk, sourced from user feedback or generated by an NL to code model. In this work, user feedback is stored in a correction data-store, associating input queries with their corrected code snippets. The KNN technique is then used to find similar queries for each query chunk. For demonstration purposes, we implemented a prototype comprising an NL to code model, a data-store for collecting user corrections, and a core component for code assembly.

Our evaluation results showcase competitive performance of *One-shot Correction* compared to other models, despite the encounter of challenges in NLP tasks. The proposed methodology also enables thorough investigation of unexpected results through straightforward approaches and promotes insights for possible enhancements. In addition, we clarified that user feedback significantly strengthens code translation models without re-training.

Ultimately, similar to the prior research contributions, this methodology also directly tackles the *programming barrier* </> while broadening libraries for code generation tasks, mitigating the *reuse problem* ♻. In the following chapter, we briefly discuss potential improvements for our proposed approaches and outline future research directions aligned with this dissertation work.

Future Work

CHAPTER 8

This chapter concludes the dissertation by outlining suggested enhancements for our proposed methodologies and delineating future research directions in alignment with our work.

8.1 Improvements for Proposed Approaches

Before delving into future research directions, we recall possible refinements for our contributions, discussed in Chapters 3–6.

NLDSL extension. Firstly, although the *NLDSL* extension exhibits its utility in resolving certain shortcomings of *low-code* approaches (e.g. lack of customization and *vendor lock-in*), there is still room for development. For example, integrating a user-friendly setting page into IDEs would facilitate parameter setup for users. Furthermore, a centralized forum for users to share customized DSLs would improve DSL accessibility.

Additionally, enhancing the code completion feature by suggesting DSL grammars alongside subsequent DSL tokens would be beneficial, reducing the necessity to memorize DSL grammars. Moreover, a user perspective survey should be conducted to gather insights for future practical features. Further suggestions are provided in Chapter 3, Section 3.5.2.

Extended Network. For our second contribution (detailed in Chapter 4), the *Extended Network model* is adaptable to integrate diverse models ranging from classical to neural ML-based approaches. Therefore, potential extensions encompass various *n-gram* models for classical methods and enhanced code completion models (e.g. Transformer-based architectures) for neural networks. Besides, expanding ensemble sizes, optimizing model hierarchies (i.e. model selection mechanism), and fine-tuning neural network architectures are also promising directions.

Code Token Type Taxonomy. Subsequently, despite the possibility to employ our refined evaluation (presented in Chapter 5) on various Python datasets, developing the methodology to include other programming languages would reinforce its versatility. Furthermore, information of code token types retrieved from our *CT3* could provide valuable hints for code completion. Consequently, an implementation of specialized code predictors that incorporate these hints may improve the outcomes.

One-shot Correction. Ultimately, even though our proposed *One-shot Correction* strategy effectively integrates user feedback into generative AI models without re-training, extending the approach on large datasets and multi-user systems would further demonstrate its utility. This involves upgrading the underlying structure of the correction data-store to efficiently handle query-correction pairs for numerous users and corrections. Dynamic Sparse Distributed Memory could serve as a viable solution for this purpose.

Additionally, lessening assumptions in code building should be considered. Particularly, rules for identifier renaming, parameter determination and input query format should be refined to adapt more intricate use cases. Finally, adopting a flexible rule selection mechanism based on input queries and contextual factors would be advantageous. Further details are outlined in Chapter 6, Section 6.6.2.

8.2 Potential Future Research Directions

Besides our core interests on *low-code* techniques, code completion approaches, and NL to code models (as mentioned in Chapter 2), there are alternative directions to tackle the challenges posed by data science-related tasks. Furthermore, the remarkable development of AI and ML also fosters new approaches and models, which are presented below as potential augmentations aligning with our work.

8.2.1 Code to Code Translation

Anaconda surveys from 2020 to 2022 reveal that practitioners in data science encounter the roadblock of switching between Programming Languages (PLs) when moving their data science models to production environments (detailed in Appendix A.3). Code-to-code translation or cross-PL translation could be a possible research direction for this matter, as noted in Section 2.2.1 of Chapter 2.

Code-to-code translation involves converting code from one PL to another while retaining as many original features as possible (Dehaerne et al., 2022). Traditionally, these tools tokenize the input source code and generate a corresponding AST, subsequently utilizing handcrafted rewrite rules for the translation step. Nevertheless, creating these rules is laborious and demands expertise in both the source and target languages (Roziere et al., 2020).

Multiple approaches have been explored to address this challenge, ranging from supervised models like tree-to-tree neural networks (Chen et al., 2018), unsupervised models such as TransCoder (Roziere et al., 2020) and kNN-ECD (Xue et al., 2023), to recent Large Language Models (LLMs) like CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023b), GPT-4 from OpenAI, and Llama 2 (Touvron et al., 2023b).

A recent study of Pan et al. (2024) highlights the current limitations in reliably utilizing LLMs for automating code translation. Various bug types were unveiled, such as syntactic, semantic, dependency, and data-related bugs. The study identifies the major challenge of fitting source language code into the constrained context window of LLMs. Pan et al. also proposed potential advancements for LLMs in code translation, including (i) integrating auxiliary information like inter-file dependencies, variable declarations, and function signatures; (ii) creating prompts that build upon each other; and (iii) fine-tuning LLMs to target specific translation bugs.

8.2.2 Knowledge-enhanced Large Language Models

LLMs show promise for various downstream NLP tasks, yet they face constraints in accessing current information and accurately manipulating knowledge, resulting in fact hallucination.

These limitations impede performance on knowledge-centric tasks (Lewis et al., 2020; Schick et al., 2024). Diverse techniques have been proposed to resolve this challenge. We outline some notable ones as follows:

Retrieval-augmented code generation. Initially, Lewis et al. (2020) introduced Retrieval-Augmented Generation (RAG) for NLP tasks, which comprises a combination of pre-trained parametic and non-parametic (i.e. retrieval-based) memories. This concept has been extended to source code, as exemplified by Parvez et al. (2021), Lu et al. (2022), and Choi et al. (2023). Further improvements, suggested by Zhao et al. (2023), include retrieval-augmented multimodal reasoning, building a multimodal knowledge index, and pre-training multimodal retrieval instead of fine-tuning pre-trained models.

Prompting techniques. Subsequently, motivated by the potential of LLMs in in-context few-shot learning via prompting (Brown et al., 2020), various prompting techniques have been applied to enhance the reasoning ability of LLMs. Prominent methods encompass plan-and-solve prompting (Wang et al., 2023a), chain-of-thought (Wei et al., 2022), tree of thoughts (Yao et al., 2024), everything of thoughts (Ding et al., 2023), and Multistage Bug Fixer (Weng et al., 2023). The latter adapted tree of thoughts for automated program repair. Specifically, to deal with the challenge of lengthy prompts, which exceed LLM constraints, Jiang et al. (2023b) presented LLMLingua, a compression method shortening these prompts while preserving semantic integrity.

External tool utilization. An alternative strategy to enhance LLMs involves guiding their handling of external tools (e.g. search engines, web browsers, or Python interpreters). This approach has been investigated in several studies, such as internet-augmented (Komeili et al., 2022) and Toolformer (Schick et al., 2024), with benchmarks like UltraTool (Huang et al., 2024), MINT (Wang et al., 2023c), and API-Bank (Li et al., 2023a).

In line with this trend, OpenAI (2023) has integrated function calling into their API updates since GPT-4-0613 and GPT-3.5-turbo-0613 models. Notably, Wu (2024) announced Devin, which is claimed as "the world's first fully autonomous AI software engineer". Devin provides essential developer tools such as a shell, a code editor, and a browser within a sandboxed compute environment. Testing on a subset of SWE-bench dataset (Jimenez et al., 2023) demonstrates Devin's superior performance over GPT-4 and SWE-Llama-13B (i.e. fine-tuned CodeLlama for SWE-bench). We expect that this direction will gain increased attention in the near future.

8.2.3 Addressing Transformers' Shortcomings

Transformers with their self-attention mechanism represent a significant advancement for LLMs, yet encounter certain drawbacks. Apart from the challenge of incorporating new data without re-training, as detailed in Chapter 6, we outline below further limitations of Transformers and corresponding prominent solutions.

Model size and computational cost. To begin with, increasing the scale of Transformers or LLMs offers various advantages (Wei et al., 2022). However, such scaling typically entails larger model sizes, leading to increased computational costs and inference latency. As a result, achieving a balance between high performance and efficiency is an essential direction. In this

context, the approach of Mistral 7B (Jiang et al., 2023a) and its successor, Mixtral 8x7B (Jiang et al., 2024), is a noteworthy consideration.

Mistral utilizes grouped-query attention and sliding window attention for accelerated inference and reduced cost. Meanwhile, Mixtral, resembling Mistral's architecture, replaces feed-forward blocks with sparse mixture-of-experts layers. This strategy increases model parameters while managing costs and latency by using only a subset of parameters per token. Mixtral outperforms Llama 2 70B across mathematics, code generation, and multilingual benchmarks.

Inference on long sequences. Additionally, although Transformers excel in accelerating the training process, they face a challenge when predicting long sequences. This problem arises from the need to recalculate attention for the entire sequence to generate subsequent tokens (Grootendorst, 2024). Consequently, Transformers suffer from inefficient inference for sequences exceeding ten thousand tokens (Gu et al., 2023).

An alternative method for handling long-range dependencies is Linear State-Space Layer (LSSL), presented by Gu et al. (2021b), which integrates the strengths of RNN, CNN, and Continuous-time models, based on State Space Model (SSM). While LSSL theoretically tackles long-range dependencies, practical implementation is hindered by prohibitive computational and memory demands. Gu et al. (2021a) later introduced Structured State Space sequence model (S4) as an augmentation to LSSL, establishing a state-of-the-art for attention-free models with fast generation ability.

Nevertheless, S4 is restricted in efficiently selecting data based on input (i.e. focusing on or ignoring specific inputs). Gu et al. (2023) then proposed Mamba, a selective state space model, to address this limitation through a selection mechanism. Mamba is claimed as the first linear-time sequence model achieving Transformer-level performance, with rapid training and inference (5x higher throughput than Transformers). Furthermore, Mamba can support real data sequences up to a million tokens in length.

Continual learning. Ultimately, Transformers, like many Deep Learning (DL) models, experience the issue of catastrophic forgetting during continual learning, where old memories are overwritten by new ones as learning progresses sequentially (Pourcel et al., 2022). An intriguing solution suggested by Shen et al. (2023) preserves associations between representation neurons and output classes in a class-incremental learning framework. Shen et al. demonstrated the method through their FlyModel, an associative continual learning algorithm.

This algorithm draws inspiration from odor-behavior associations of fruit flies, with a two-layer neural circuit, namely sparse coding and perception-like associative learning. Sparse coding activates distinct neuron populations for different odors, reducing memory interference, while associative learning selectively adjusts synapses between odor-activated neurons and associated output neurons to prevent overwriting unrelated memory. The FlyModel surpasses other neural-inspired algorithms in mitigating catastrophic forgetting. However, applying this approach to Transformers or other DL models requires further investigation.

Part V

References

List of Acronyms

AI Artifical Intelligence . . . 3

ANN Artificial Neural Network . . . 35

API Application Programming Interface . . . 4

AST Abstract Syntax Tree . . . 9

BPE Byte-Pair Encoding . . . 115

CD Continuous Deployment . . . 16

CFG Context-Free Grammar . . . 31

CI Continuous Integration . . . 16

CNN Convolutional Neural Network . . . 29

CSS Cascading Style Sheets . . . 20

CSV Comma-separated Values . . . 57

CT3 Code Token Type Taxonomy . . . 9

DFS Depth-First Search . . . 34

DL Deep Learning . . . 29

DSL Domain-Specific Language . . . 4

EBNF Extended Backus-Naur Form . . . 59

FFNN Feed-forward Neural Network . . . 35

GNN Graph Neural Network . . . 29

GPL General-purpose Programming Language . . . 6

GRU Gated Recurrent Unit . . . 29

GUI Graphical User Interface . . . 10

HOG Higher Order Grammar 33

HTML HyperText Markup Language 20

IDE Integrated Development Environment 6

JSON JavaScript Object Notation 71

KNN k-Nearest Neighbor 29

LLM Large Language Model 18

LSP Language Server Protocol 9

LSTM Long Short-Term Memory 29

ML Machine Learning 3

MLM Masked Language Model 50

MLP Multi-layer Perceptron 43

NL Natural Language 5

NLDSL Natural Language to Domain Specific Language 55

NLP Natural Language Processing 30

OOV Out-of-Vocabulary 39

OS Operating System 73

PBNL Programming by Natural Language 18

PCFG Probabilistic Context Free Grammars 32

PHOG Probabilistic Higher Order Grammar 33

PL Programming Language 15

RNN Recurrent Neural Network 29

SQL Structured Query Language 20

UML Unified Modeling Language 20

VSCode Visual Studio Code 28

XML Extensible Markup Language 20

YAML Yet Another Markup Language 75

Bibliography

3Blue1Brown (2017). *Neural Networks*. URL: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. (Accessed: 05 February 2024).

Abiodun, O. I., Jantan, A., Omolara, A. E., Dada, K. V., Mohamed, N. A., and Arshad, H. (2018). "State-of-the-art in artificial neural network applications: A survey". In: *Heliyon* 4.11.

Abney, S. P. (1992). "Parsing by chunks". In: *Principle-based parsing: Computation and Psycholinguistics*, pp. 257–278.

Acciarini, C., Cappa, F., Boccardelli, P., and Oriani, R. (2023). "How can organizations leverage big data to innovate their business models? A systematic literature review". In: *Technovation* 123, p. 102713.

Agadumo, J. (2023). *Top 10 Data Preparation Tools Revolutionizing the Analytics Landscape in 2023*. URL: https://www.datameer.com/blog/top-10-data-preparation-tools-revolutionizing-the-analytics-landscape-in-2023/. (Accessed: 22 January 2024).

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. (2021). "Unified pre-training for program understanding and generation". In: *arXiv preprint arXiv:2103.06333*.

AIP Group (2022a). *AIP Webpage for DSLs Creation with NLDSL Visual Studio Code Extension*. URL: https://aip.ifi.uni-heidelberg.de/software/nldsl/custom-dsl-creation. (Accessed: 01 March 2024).

AIP Group (2022b). *AIP Webpage for NLDSL Visual Studio Code Extension*. URL: https://aip.ifi.uni-heidelberg.de/software/nldsl. (Accessed: 01 March 2024).

Alammar, J. (2018a). *The Illustrated Transformer*. URL: https://jalammar.github.io/illustrated-transformer/. (Accessed: 12 February 2024).

Alammar, J. (2018b). *Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)*. URL: https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/. (Accessed: 12 February 2024).

Alharthi, A., Krotov, V., and Bowman, M. (2017). "Addressing barriers to big data". In: *Business Horizons* 60.3, pp. 285–292.

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2018). "A Survey of Machine Learning for Big Code and Naturalness". In: *ACM Comput. Surv.* 51.4, 81:1–81:37. DOI: 10.1145/3212695. URL: http://doi.acm.org/10.1145/3212695.

Alon, U., Sadaka, R., Levy, O., and Yahav, E. (2020a). "Structural Language Models of Code". In: *arXiv:1910.00577*. URL: http://arxiv.org/abs/1910.00577 (visited on February 28, 2020).

Alon, U., Sadaka, R., Levy, O., and Yahav, E. (2020b). "Structural language models of code". In: *International conference on machine learning*. tex.organization: PMLR, 245–256.

Alves, I. R. (2023). *Making your life easier with domain-specific languages (DSLs)*. URL: https://medium.com/wearewaes/making-your-life-easier-with-domain-specific-languages-dsl-1838d351d35. (Accessed: 10 January 2024).

Ammirato, S., Felicetti, A. M., Linzalone, R., Corvello, V., and Kumar, S. (2023). "Still our most important asset: A systematic review on human resource management in the midst of the fourth industrial revolution". In: *Journal of Innovation & Knowledge* 8.3, p. 100403.

Anaconda (2018). *State of Data Science Report*. URL: https://know.anaconda.com/rs/387-XNW-688/images/2018-06-Anaconda-State-of-data-science-report.pdf. (Accessed: 17 November 2023).

Anaconda (2020). *State of Data Science: Moving from hype toward maturity*. URL: https://know.anaconda.com/rs/387-XNW-688/images/Anaconda-SODS-Report-2020-Final.pdf. (Accessed: 17 November 2023).

Anaconda (2021). *State of Data Science: On the path to impact*. URL: https://know.anaconda.com/rs/387-XNW-688/images/Anaconda-2021-SODS-Report-Final.pdf. (Accessed: 17 November 2023).

Anaconda (2022). *State of Data Science: Paving the way for innovation*. URL: https://www.anaconda.com/resources/whitepapers/state-of-data-science-report-2022. (Accessed: 17 November 2023).

Anaconda (2023). *State of Data Science: AI takes center stage*. URL: https://www.anaconda.com/state-of-data-science-report-2023. (Accessed: 17 November 2023).

Anderson, B. M. (2021). *These Are the Fastest-Growing Jobs Around the World*. URL: https://www.linkedin.com/business/talent/blog/talent-strategy/jobs-on-the-rise-list. (Accessed: 17 November 2023).

Anderson, C. (2015). *Creating a data-driven organization: Practical advice from the trenches*. O'Reilly Media, Inc.

Andrzejak, A., Kiefer, K., Costa, D. E., and Wenz, O. (2019a). "Agile construction of data science DSLs (tool demo)". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 27–33.

Andrzejak, A., Wenz, O., and Costa, D. (2019b). "One DSL to Rule Them All: IDE-Assisted Code Generation for Agile Data Analysis". In: *arXiv preprint arXiv:1904.09818*. DOI: https://doi.org/10.48550/arXiv.1904.09818.

Appen (2019). *The State of AI and Machine Learning*. URL: https://visit.appen.com/WC-2019-State-of-AI-Report-LP.html?utm_source=Web&utm_medium=ResourceCenter. (Accessed: 20 November 2023).

Asare, O., Nagappan, M., and Asokan, N (2022). "Is github's copilot as bad as humans at introducing vulnerabilities in code?" In: *arXiv preprint arXiv:2204.04741*.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). "Layer normalization". In: *arXiv preprint arXiv:1607.06450*.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. URL: http://arxiv.org/abs/1409.0473.

BARC (2015). *Insufficient Skills Are Curbing the Big Data Boom*. URL: https://bi-survey.com/challenges-big-data-analytics. (Accessed: 21 November 2023).

Barke, S., James, M. B., and Polikarpova, N. (2023). "Grounded copilot: How programmers interact with code-generating models". In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1, pp. 85–111.

Bayer, J. S. (2015). "Learning sequence representations". PhD thesis. Technische Universität München.

Bengio, Y., Ducharme, R., and Vincent, P. (2000). "A neural probabilistic language model". In: *Advances in neural information processing systems* 13.

Bengio, Y., Goodfellow, I., and Courville, A. (2017). *Deep learning*. Vol. 1. MIT press Cambridge, MA, USA.

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Bielik, P., Raychev, V., and Vechev, M. (2016). "PHOG: Probabilistic Model for Code". In: *Proceedings of The 33rd International Conference on Machine Learning*. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, pp. 2933–2942. URL: http://proceedings.mlr.press/v48/bielik16.html.

Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., and Gazit, I. (2022). "Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools". In: *Queue* 20.6, pp. 35–57.

Bock, A. C. and Frank, U. (2021). "Low-code platform". In: *Business & Information Systems Engineering* 63, pp. 733–740.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. (2019). "Generative Code Modeling with Graphs". In: *International Conference on Learning Representations (ICLR 2019)*. URL: http://arxiv.org/abs/1805.08490.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). "Language models are few-shot learners". In: *Advances in neural information processing systems* 33, pp. 1877–1901.

Cai, Y., Mao, S., Wu, W., Wang, Z., Liang, Y., Ge, T., Wu, C., You, W., Song, T., Xia, Y., et al. (2023). "Low-code LLM: Visual Programming over LLMs". In: *arXiv preprint arXiv:2304.08103*.

Cambronero, J., Gulwani, S., Le, V., Perelman, D., Radhakrishna, A., Simon, C., and Tiwari, A. (2023). "FlashFill++: Scaling programming by example by cutting to the chase". In: *Proceedings of the ACM on Programming Languages* 7.POPL, pp. 952–981.

Campagne, F. (2014). *The MPS language workbench: volume I*. Vol. 1. Fabien Campagne.

Carpenter, A. (2021). *7 Tips to Help You Learn a New Programming Language Fast*. URL: https://www.codecademy.com/resources/blog/how-to-learn-a-new-programming-language-fast/. (Accessed: 22 June 2023).

Charitsis, C., Piech, C., and Mitchell, J. C. (2022). "Using nlp to quantify program decomposition in cs1". In: *Proceedings of the Ninth ACM Conference on Learning@ Scale*, pp. 113–120.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374*.

Chen, S. F. and Goodman, J. (1999). "An empirical study of smoothing techniques for language modeling". In: *Computer Speech & Language* 13.4, pp. 359–394.

Chen, X., Liu, C., and Song, D. (2018). "Tree-to-tree neural networks for program translation". In: *Advances in neural information processing systems* 31.

Chirkova, N. and Troshin, S. (2020). "A simple approach for handling out-of-vocabulary identifiers in deep learning for source code". In: *arXiv preprint arXiv:2010.12663*.

Chirkova, N. and Troshin, S. (2021). "Empirical study of transformers for source code". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 703–715.

Choi, Y., Na, C., Kim, H., and Lee, J.-H. (2023). "READSUM: Retrieval-Augmented Adaptive Transformer for Source Code Summarization". In: *IEEE Access*.

Ciniselli, M., Cooper, N., Pascarella, L., Mastropaolo, A., Aghajani, E., Poshyvanyk, D., Di Penta, M., and Bavota, G. (2021a). "An Empirical Study on the Usage of Transformer Models for Code Completion". In: *IEEE Transactions on Software Engineering*.

Ciniselli, M., Cooper, N., Pascarella, L., Poshyvanyk, D., Di Penta, M., and Bavota, G. (2021b). "An Empirical Study on the Usage of BERT Models for Code Completion". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 108–119. DOI: 10.1109/MSR52588.2021.00024.

Córdoba-Sánchez, I. and De Lara, J. (2016). "Ann: A domain-specific language for the effective design and validation of Java annotations". In: *Computer Languages, Systems & Structures* 45, pp. 164–190.

CrowdFlower (2015). *Data Scientist Report*. URL: https://visit.figure-eight.com/2015-data-scientist-report. (Accessed: 17 November 2023).

CrowdFlower (2016). *Data Science Report*. URL: https://visit.figure-eight.com/rs/416-ZBE-142/images/CrowdFlower_DataScienceReport_2016.pdf. (Accessed: 17 November 2023).

CrowdFlower (2017). *Data Scientist Report*. URL: https://visit.figure-eight.com/rs/416-ZBE-142/images/CrowdFlower_DataScienceReport.pdf. (Accessed: 17 November 2023).

Dai, D., Dong, L., Hao, Y., Sui, Z., Chang, B., and Wei, F. (2022). "Knowledge Neurons in Pretrained Transformers". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8493–8502.

Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., and Jiang, Z. M. J. (2023). "Github copilot ai pair programmer: Asset or liability?" In: *Journal of Systems and Software* 203, p. 111734.

Databricks (2023). *State of Data + AI*. URL: https://www.databricks.com/discover/state-of-data-ai. (Accessed: 22 November 2023).

Davenport, T. H. and Patil, D. (2012). *Data Scientist: The Sexiest Job of the 21st Century*. URL: https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century. (Accessed: 31 October 2023).

Davenport, T. H. and Patil, D. (2022). *Is Data Scientist Still the Sexiest Job of the 21st Century?* URL: https://hbr.org/2022/07/is-data-scientist-still-the-sexiest-job-of-the-21st-century. (Accessed: 31 October 2023).

Dearmer, A. (2023). *Top 12 Data Preparation Tools*. URL: https://www.integrate.io/blog/top-data-preparation-tools/. (Accessed: 22 January 2024).

Dehaerne, E., Dey, B., Halder, S., De Gendt, S., and Meert, W. (2022). "Code generation using machine learning: A systematic review". In: *IEEE Access* 10, pp. 82434 –82455.

Dejanović, I., Dejanović, M., Vidaković, J., and Nikolić, S. (2021). "PyFlies: A Domain-Specific Language for Designing Experiments in Psychology". In: *Applied Sciences* 11.17, p. 7823.

Dejanović, I., Vaderna, R., Milosavljević, G., and Vuković, Ž. (2017). "Textx: a python tool for domain-specific languages implementation". In: *Knowledge-based systems* 115, pp. 1–4.

DeLine, R. A. (2021). "Glinda: Supporting data science with live programming, GUIs and a Domain-specific Language". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–11.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). "BERT: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805*.

Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., and Ziane, M. (2012). "Robotml, a domain-specific language to design, simulate and deploy robotic applications". In: *Simulation, Modeling, and Programming for Autonomous Robots: Third International Conference, SIMPAR 2012, Tsukuba, Japan, November 5-8, 2012. Proceedings 3*. Springer, pp. 149–160.

Di Ruscio, D., Kolovos, D., Lara, J. de, Pierantonio, A., Tisi, M., and Wimmer, M. (2022). "Low-code development and model-driven engineering: Two sides of the same coin?" In: *Software and Systems Modeling* 21.2, pp. 437–446.

Ding, R., Zhang, C., Wang, L., Xu, Y., Ma, M., Zhang, W., Qin, S., Rajmohan, S., Lin, Q., and Zhang, D. (2023). "Everything of thoughts: Defying the law of penrose triangle for thought generation". In: *arXiv preprint arXiv:2311.04254*.

Ding, Y., Buratti, L., Pujar, S., Morari, A., Ray, B., and Chakraborty, S. (2021). "Contrastive Learning for Source Code with Structural and Functional Properties". In: *arXiv preprint arXiv:2110.03868*.

Doshi, K. (2021). *Transformers Explained Visually (Part 3): Multi-head Attention, deep dive*. URL: https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive-1c1ff1024853. (Accessed: 12 February 2024).

Drosos, I., Barik, T., Guo, P. J., DeLine, R., and Gulwani, S. (2020). "Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists". In: *Proceedings of the 2020 CHI conference on human factors in computing systems*, pp. 1–12.

Egidi, M. (2006). "Decomposition patterns in problem solving". In: *Contributions to Economic Analysis* 280, pp. 15–46.

Elgohary, A., Meek, C., Richardson, M., Fourney, A., Ramos, G., and Awadallah, A. H. (2021). "NL-EDIT: Correcting semantic parse errors through natural language interaction". In: *arXiv preprint arXiv:2103.14540*.

Elshan, E., Dickhaut, E., and Ebel, P. A. (2023). "An investigation of why low code platforms provide answers and new challenges". In: *Proceedings of the 56th Hawaii International Conference on System Sciences*, pp. 6159–6168.

Erdweg, S., Van Der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., et al. (2015). "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Computer Languages, Systems & Structures* 44, pp. 24–47.

Evtikhiev, M., Bogomolov, E., Sokolov, Y., and Bryksin, T. (2023). "Out of the bleu: how should we assess quality of the code generation models?" In: *Journal of Systems and Software* 203, p. 111741.

Fan, A., Gardent, C., Braud, C., and Bordes, A. (2021). "Augmenting transformers with KNN-based composite memory for dialog". In: *Transactions of the Association for Computational Linguistics* 9, pp. 82–99.

Felleisen, M. (1990). "On the expressive power of programming languages". In: *European Symposium on Programming*. Springer, pp. 134–151.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155*.

FigureEight (2018). *Data Scientist Report*. URL: https://visit.figure-eight.com/rs/416-ZBE-142/images/Data-Scientist-Report.pdf. (Accessed: 20 November 2023).

Foote, K. D. (2021). *A Brief History of Data Science*. URL: https://www.dataversity.net/brief-history-data-science/. (Accessed: 18 December 2023).

Fowler, M. (2010). *Domain-Specific languages*. Addison-Wesley Professional.

Gafner, J. (2023). *Best Jobs of 2023*. URL: https://www.indeed.com/career-advice/news/best-jobs-of-2023. (Accessed: 31 October 2023).

Gal, Y. and Ghahramani, Z. (2016). "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., pp. 1019–1027. URL: http://papers.nips.cc/paper/6241-a-theoretically-grounded-application-of-dropout-in-recurrent-neural-networks.pdf.

Geva, M., Schuster, R., Berant, J., and Levy, O. (2021). "Transformer Feed-Forward Layers Are Key-Value Memories". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 5484–5495.

Gozalo-Brizuela, R. and Garrido-Merchan, E. C. (2023). "ChatGPT is not all you need. A State of the Art Review of large Generative AI models". In: *arXiv preprint arXiv:2301.04655*.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). "Speech recognition with deep recurrent neural networks". In: *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, pp. 6645–6649.

Grootendorst, M. (2024). *A Visual Guide to Mamba and State Space Models*. URL: https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-mamba-and-state. (Accessed: 31 March 2024).

Gu, A. and Dao, T. (2023). "Mamba: Linear-time sequence modeling with selective state spaces". In: *arXiv preprint arXiv:2312.00752*.

Gu, A., Goel, K., and Ré, C. (2021a). "Efficiently modeling long sequences with structured state spaces". In: *arXiv preprint arXiv:2111.00396*.

Gu, A., Johnson, I., Goel, K., Saab, K., Dao, T., Rudra, A., and Ré, C. (2021b). "Combining recurrent, convolutional, and continuous-time models with linear state space layers". In: *Advances in neural information processing systems* 34, pp. 572–585.

Gulwani, S. (2011). "Automating string processing in spreadsheets using input-output examples". In: *ACM Sigplan Notices* 46.1, pp. 317–330.

Gür, I., Yavuz, S., Su, Y., and Yan, X. (2018). "Dialsql: Dialogue based structured query generation". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1339–1349.

Gvero, T. and Kuncak, V. (2015). "Synthesizing Java expressions from free-form queries". In: *Proceedings of the 2015 acm sigplan international conference on object-oriented programming, systems, languages, and applications*, pp. 416–432.

Haan, K. (2024). *The Best Data Analytics Tools Of 2024*. URL: https://www.forbes.com/advisor/business/software/best-data-analytics-tools/. (Accessed: 11 January 2024).

He, K., Zhang, X., Ren, S., and Sun, J. (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Heer, J., Hellerstein, J. M., and Kandel, S. (2015). "Predictive Interaction for Data Transformation." In: *CIDR*.

Hellendoorn, V. J., Proksch, S., Gall, H. C., and Bacchelli, A. (2019). "When code completion fails: A case study on real-world completions". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, pp. 960–970.

Heyman, G., Huysegems, R., Justen, P., and Van Cutsem, T. (2021). "Natural language-guided programming". In: *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 39–55.

Hilgers, L. (2022). *The Fastest-Growing Jobs Around the World*. URL: https://www.linkedin.com/business/talent/blog/talent-strategy/fastest-growing-jobs-global. (Accessed: 17 November 2023).

Hilgers, L. (2023). *The Fastest-Growing Jobs Around the World in 2023*. URL: https://www.linkedin.com/business/talent/blog/talent-acquisition/fastest-growing-jobs-2023. (Accessed: 17 November 2023).

Hindle, A., Barr, E. T., Gabel, M., Su, Z., and Devanbu, P. (2016). "On the naturalness of software". In: *Communications of the ACM* 59.5, pp. 122–131.

Hirzel, M. (2023). "Low-code programming models". In: *Communications of the ACM* 66.10, pp. 76–85.

Hochreiter, S. and Schmidhuber, J. (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

Hoffman, K. L., Padberg, M., Rinaldi, G., et al. (2013). "Traveling salesman problem". In: *Encyclopedia of operations research and management science* 1, pp. 1573–1578.

Huang, S., Zhong, W., Lu, J., Zhu, Q., Gao, J., Liu, W., Hou, Y., Zeng, X., Wang, Y., Shang, L., et al. (2024). "Planning, Creation, Usage: Benchmarking LLMs for Comprehensive Tool Utilization in Real-World Complex Scenarios". In: *arXiv preprint arXiv:2401.17167*.

Hussain, Y., Huang, Z., and Zhou, Y. (2021). "Improving source code suggestion with code embedding and enhanced convolutional long short-term memory". In: *IET Software* 15.3, pp. 199–213.

Hussain, Y., Huang, Z., Zhou, Y., and Wang, S. (2020). "CodeGRU: Context-aware deep learning with gated recurrent unit for source code modeling". In: *Information and Software Technology* 125, p. 106309.

Imai, S. (2022). "Is GitHub copilot a substitute for human pair-programming? An empirical study". In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 319–321.

Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., and Zettlemoyer, L. (2017). "Learning a neural semantic parser from user feedback". In: *arXiv preprint arXiv:1704.08760*.

Izadi, M., Gismondi, R., and Gousios, G. (2022). "Codefill: Multi-token code completion by jointly learning from structure and naming sequences". In: *Proceedings of the 44th International Conference on Software Engineering*. ISBN: 978-1-4503-9221-1. New York, NY, USA: Association for Computing Machinery, pp. 401–412. DOI: 10.1145/3510003.3510172. URL: https://doi.org/10.1145/3510003.3510172.

Jaimovitch-López, G., Ferri, C., Hernández-Orallo, J., Martínez-Plumed, F., and Ramírez-Quintana, M. J. (2023). "Can language models automate data wrangling?" In: *Machine Learning* 112.6, pp. 2053–2082.

Janiesch, C., Zschech, P., and Heinrich, K. (2021). "Machine learning and deep learning". In: *Electronic Markets* 31.3, pp. 685–695.

JetBrains (2017). *Voice Menu - a concrete example of MPS*. URL: https://youtu.be/pVIywLXDuRo?si=UT2oDcJyryjFwPHV. (Accessed: 07 January 2024).

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. (2023a). "Mistral 7B". In: *arXiv preprint arXiv:2310.06825*.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B., Bressand, F., et al. (2024). "Mixtral of experts". In: *arXiv preprint arXiv:2401.04088*.

Jiang, H., Wu, Q., Lin, C.-Y., Yang, Y., and Qiu, L. (2023b). "Llmlingua: Compressing prompts for accelerated inference of large language models". In: *arXiv preprint arXiv:2310.05736*.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. (2023). "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *The Twelfth International Conference on Learning Representations*.

Johannessen, C. and Davenport, T. (2021). *When Low-Code/No-Code Development Works - and When It Doesn't*. URL: https://hbr.org/2021/06/when-low-code-no-code-development-works-and-when-it-doesnt. (Accessed: 02 January 2024).

Jurafsky, D. and Martin, J. H. (2008). "Speech and Language Processing: An introduction to speech recognition, computational linguistics and natural language processing". In: *Upper Saddle River, NJ: Prentice Hall*.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. (2020). "Learning and evaluating contextual embedding of source code". In: *International Conference on Machine Learning*. PMLR, pp. 5110–5121.

Kandel, S., Paepcke, A., Hellerstein, J., and Heer, J. (2011). "Wrangler: Interactive visual specification of data transformation scripts". In: *Proceedings of the sigchi conference on human factors in computing systems*, pp. 3363–3372.

Kanerva, P. (1992). *Sparse distributed memory and related models*. Tech. rep.

Karampatsis, R.-M., Babii, H., Robbes, R., Sutton, C., and Janes, A. (2020). "Big code!= big vocabulary: Open-vocabulary models for source code". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, pp. 1073–1085.

Karl, T. (2023). *The Power of No-Code and Low-Code Data Analysis Tools in Building a Data-Driven Decision-Making Culture*. URL: https://www.newhorizons.com/resources/blog/no-code-low-code-data-analysis-tools. (Accessed: 21 December 2023).

Kats, L. C. and Visser, E. (2010). "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 444–463.

Kelleher, J. D. (2019). *Deep learning*. MIT press.

Khandelwal, U., Fan, A., Jurafsky, D., Zettlemoyer, L., and Lewis, M. (2020). "Nearest neighbor machine translation". In: *arXiv preprint arXiv:2010.00710*.

Kim, S., Zhao, J., Tian, Y., and Chandra, S. (2021). "Code prediction by feeding trees to transformers". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, pp. 150–162.

Kingma, D. P. and Ba, J. (2015). "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. URL: http://arxiv.org/abs/1412.6980.

Knime (2023). *Decoding the Data Universe: State of Data Science and Machine Learning*. URL: https://info.knime.com/state-of-data-science-and-machine-learning. (Accessed: 21 November 2023).

Komeili, M., Shuster, K., and Weston, J. (2022). "Internet-Augmented Dialogue Generation". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8460–8478.

Korz, N. and Andrzejak, A. (2023). "Virtual Domain Specific Languages via Embedded Projectional Editing". In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 122–137.

Kosar, T., Martı, P. E., Barrientos, P. A., Mernik, M., et al. (2008). "A preliminary study on various implementation approaches of domain-specific language". In: *Information and software technology* 50.5, pp. 390–405.

Lakin, M. R. and Phillips, A. (2020). "Domain-specific programming languages for computational nucleic acid systems". In: *ACS Synthetic Biology* 9.7, pp. 1499–1513.

Le, K. T. and Andrzejak, A. (2024). "Rethinking AI Code Generation: A One-shot Correction Approach Based on User Feedback". In: *Automated Software Engineering* 31.60. DOI: 10.1007/s10515-024-00451-y.

Le, K. T., Rashidi, G., and Andrzejak, A. (2023). "A methodology for refined evaluation of neural code completion approaches". In: *Data Mining and Knowledge Discovery* 37.1, pp. 167–204. DOI: 10.1007/s10618-022-00866-9.

Le, T. H. M., Chen, H., and Babar, M. A. (2020). "Deep learning for source code modeling and generation: Models, applications, and challenges". In: *ACM Computing Surveys (CSUR)* 53.3, pp. 1–38.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). "Retrieval-augmented generation for knowledge-intensive nlp tasks". In: *Advances in Neural Information Processing Systems* 33, pp. 9459–9474.

Li, J., Wang, Y., Lyu, M. R., and King, I. (2018). "Code Completion with Neural Attention and Pointer Networks". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.* ISBN: 978-0-9992411-2-7. ijcai.org, pp. 4159–4165. DOI: 10.24963/ijcai.2018/578. URL: https://doi.org/10.24963/ijcai.2018/578.

Li, M., Zhao, Y., Yu, B., Song, F., Li, H., Yu, H., Li, Z., Huang, F., and Li, Y. (2023a). "API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. (2023b). "Starcoder: may the source be with you!" In: *arXiv preprint arXiv:2305.06161*.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. (2022). "Competition-level code generation with alphacode". In: *Science* 378.6624, pp. 1092–1097.

Liu, C., Wang, X., Shin, R., Gonzalez, J. E., and Song, D. (2017). *Neural Code Completion*. URL: https://openreview.net/forum?id=rJbPBt9lg.

Liu, F., Li, G., Zhao, Y., and Jin, Z. (2020). "Multi-task Learning based Pre-trained Language Model for Code Completion". In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 473–485.

Lu, S., Duan, N., Han, H., Guo, D., Hwang, S.-w., and Svyatkovskiy, A. (2022). "ReACC: A retrieval-augmented code completion framework". In: *arXiv preprint arXiv:2203.07722*.

Luo, Y., Liang, P., Wang, C., Shahin, M., and Zhan, J. (2021). "Characteristics and challenges of low-code development: the practitioners' perspective". In: *Proceedings of the 15th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, pp. 1–11.

Luong, M.-T., Pham, H., and Manning, C. D. (2015). "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Ed. by L. Màrquez, C. Callison-Burch, and J. Su. Lisbon, Portugal: Association for Computational Linguistics, pp. 1412–1421. DOI: 10.18653/v1/D15-1166. URL: https://aclanthology.org/D15-1166.

Makonin, S., McVeigh, D., Stuerzlinger, W., Tran, K., and Popowich, F. (2016). "Mixed-initiative for big data: The intersection of human+ visual analytics+ prediction". In: *2016 49th Hawaii international conference on system sciences (HICSS)*. IEEE, pp. 1427–1436.

Managoli, G. (2020). *What developers need to know about domain-specific languages*. URL: https://opensource.com/article/20/2/domain-specific-languages. (Accessed: 07 January 2024).

Martinez, E. and Pfister, L. (2023). "Benefits and limitations of using low-code development to support digitalization in the construction industry". In: *Automation in Construction* 152, p. 104909.

Meng, K., Bau, D., Andonian, A., and Belinkov, Y. (2022). "Locating and editing factual associations in GPT". In: *Advances in Neural Information Processing Systems* 35, pp. 17359–17372.

Mernik, M., Heering, J., and Sloane, A. M. (2005). "When and how to develop domain-specific languages". In: *ACM computing surveys (CSUR)* 37.4, pp. 316–344.

Microsoft (2022). *Anatomy of a Visual Studio extension*. URL: https://learn.microsoft.com/en-us/visualstudio/extensibility/vsix/get-started/extension-anatomy. (Accessed: 06 March 2024).

Microsoft (2023). *Use Azure Pipelines*. URL: https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/pipelines-get-started. (Accessed: 06 March 2024).

Mikalef, P., Wetering, R. van de, and Krogstie, J. (2021). "Building dynamic capabilities by leveraging big data analytics: The role of organizational inertia". In: *Information & Management* 58.6, p. 103412.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems* 26.

Mikolov, T., Yih, W.-t., and Zweig, G. (2013c). "Linguistic regularities in continuous space word representations". In: *Proceedings of the 2013 conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Atlanta, Georgia: Association for Computational Linguistics, pp. 746–751.

Mohapatra, N., Sarraf, N., et al. (2021). "Domain based chunking". In: *International Journal on Natural Language Computing (IJNLC) Vol* 10.

Nadkarni, P. M., Ohno-Machado, L., and Chapman, W. W. (2011). "Natural language processing: an introduction". In: *Journal of the American Medical Informatics Association* 18.5, pp. 544–551. DOI: 10.1136/amiajnl-2011-000464. eprint: http://oup.prod.sis.lan/jamia/article-pdf/18/5/544/5962687/18-5-544.pdf. URL: https://doi.org/10.1136/amiajnl-2011-000464.

Nguyen, N. and Nadi, S. (2022). "An empirical evaluation of GitHub copilot's code suggestions". In: *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 1–5.

Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. (2013). "A statistical semantic language model for source code". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 532–542.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. (2022). "Codegen: An open large language model for code with multi-turn program synthesis". In: *arXiv preprint arXiv:2203.13474*.

OpenAI (2023). *Function calling and other API updates*. URL: https://openai.com/blog/function-calling-and-other-api-updates. (Accessed: 31 March 2024).

Pachev, A. (2007). *Understanding MySQL Internals*. O'Reilly Media, Inc.

Pan, R., Ibrahimzada, A. R., Krishna, R., Sankar, D., Wassi, L. P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., and Jabbarvand, R. (2024). "Lost in translation: A study of bugs introduced by large language models while translating code". In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, pp. 866–866.

Parvez, M. R., Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. (2021). "Retrieval augmented code generation and summarization". In: *arXiv preprint arXiv:2108.11601*.

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2013). "How to construct deep recurrent neural networks". In: *arXiv preprint arXiv:1312.6026*.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2022). "Asleep at the keyboard? assessing the security of github copilot's code contributions". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 754–768.

Penke, C. (2022). *A mathematician's introduction to transformers and large language models*. URL: https://x-dev.pages.jsc.fz-juelich.de/2022/07/13/transformers-matmul.html. (Accessed: 29 June 2024).

Petricek, T., Den Burg, G. J. van, Nazábal, A., Ceritli, T., Jiménez-Ruiz, E., and Williams, C. K. (2022). "AI Assistants: A Framework for Semi-Automated Data Wrangling". In: *IEEE Transactions on Knowledge and Data Engineering*.

Pfleger, D. (2020). "Building a wizard to support developers in generating new DSLs using the NLDSL framework". BA thesis. Heidelberg University, Germany.

Pingali, K. (2023). *Introduction to Parsing (adapted from CS 164 at Berkeley)*. URL: https://www.cs.utexas.edu/~pingali/CS380C/2023/lectures/parsing/parsingIntro.pdf. (Accessed: 01 February 2024).

Poupart, P. (2019). *CS480/680 Lecture 19: Attention and Transformer Networks*. URL: https://www.youtube.com/watch?v=OyFJWRnt_AY. (Accessed: 18 February 2024).

Pourcel, J., Vu, N.-S., and French, R. M. (2022). "Online task-free continual learning with dynamic sparse distributed memory". In: *European Conference on Computer Vision*. Springer, pp. 739–756.

Press, G. (2013). *A Very Short History Of Data Science*. URL: https://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/. (Accessed: 18 December 2023).

Provost, F. and Fawcett, T. (2013). "Data science and its relationship to big data and data-driven decision making". In: *Big data* 1.1, pp. 51–59.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). "Improving language understanding by generative pre-training". In: *OpenAI*.

Rahman, M. M., Watanobe, Y., and Nakamura, K. (2020). "A neural network based intelligent support model for program code completion". In: *Scientific Programming* 2020, pp. 1–18.

Ramshaw, L. A. and Marcus, M. P. (1999). "Text chunking using transformation-based learning". In: *Natural language processing using very large corpora*, pp. 157–176.

Rashidi, G. (2021). "Code Token Type Taxonomy for Transformer-Based Code Completion". BA thesis. Heidelberg University, Germany.

Raychev, V., Bielik, P., and Vechev, M. (2016). "Probabilistic Model for Code with Decision Trees". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. ISBN: 978-1-4503-4444-9. Amsterdam, Netherlands: ACM, pp. 731–747. DOI: 10.1145/2983990.2984041. URL: http://doi.acm.org/10.1145/2983990.2984041.

Raychev, V., Vechev, M., and Yahav, E. (2014). "Code completion with statistical language models". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 419–428.

Raza, M. and Gulwani, S. (2017). "Automated data extraction using predictive program synthesis". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1.

RedHat (2019). *What is an IDE?* URL: https://www.redhat.com/en/topics/middleware/what-is-ide. (Accessed: 27 January 2024).

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. (2020). "Codebleu: a method for automatic evaluation of code synthesis". In: *arXiv preprint arXiv:2009.10297*.

Reynolds, L. and McDonell, K. (2021). "Prompt programming for large language models: Beyond the few-shot paradigm". In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–7.

Richardson, C., Rymer, J. R., Mines, C., Cullen, A., and Whittaker, D. (2014). "New development platforms emerge for customer-facing applications". In: *Forrester: Cambridge, MA, USA* 15.

Robillard, M., Walker, R., and Zimmermann, T. (2010). "Recommendation Systems for Software Engineering". In: *IEEE Software* 27.4, pp. 80–86. DOI: 10.1109/MS.2009.161.

Roziere, B., Lachaux, M.-A., Chanussot, L., and Lample, G. (2020). "Unsupervised translation of programming languages". In: *Advances in neural information processing systems* 33, pp. 20601–20611.

Rubin, J. (2023). *DataGPT Launches out of Stealth to Help Users Talk Directly to Their Data Using Everyday Language*. URL: https://datagpt.com/blog/datagpt-launches/. (Accessed: 23 January 2024).

Sagi, O. and Rokach, L. (2018). "Ensemble learning: A survey". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8.4, e1249.

Saint-Exupéry, A. d. (2016). *The Little Prince*. Trans. by M. Herbert and Cillero & de Motta. ISBN: 978-3-7306-0420-5. Anaconda Verlag.

Sajnani, H. (2016). *Large-scale code clone detection*. University of California, Irvine.

Sanaulla, M. (2013). *Creating Internal DSLs in Java and Java 8*. URL: https://dzone.com/articles/creating-internal-dsls-java. (Accessed: 07 January 2024).

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. (2024). "Toolformer: Language models can teach themselves to use tools". In: *Advances in Neural Information Processing Systems* 36.

Schlegel, V., Lang, B., Handschuh, S., and Freitas, A. (2019). "Vajra: step-by-step programming with natural language". In: *Proceedings of the 24th International Conference on Intelligent User Interfaces*, pp. 30–39.

Schmitt, O. (2019). "Development of a Code Fragment Recommender System using the Language Server Protocol in the Context of Data Analysis with Python". BA thesis. Heidelberg University, Germany.

Schumacher, M. E. H. (2019). "Combining Deep Learning and Probabilistic Models for Intelligent Code Completion in Dynamically Typed Languages". BA thesis. Heidelberg University, Germany.

Schumacher, M. E. H., Le, K. T., and Andrzejak, A. (2020). "Improving code recommendations by combining neural and classical machine learning approaches". In: *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops*, pp. 476–482. DOI: 10.1145/3387940.3391489.

Scott, M. (2000). *Programming language pragmatics*. Morgan Kaufmann.

Sharma, R., Chen, F., Fard, F., and Lo, D. (2022). "An exploratory study on code attention in BERT". In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pp. 437–448.

Sharma, S., Sharma, S., and Athaiya, A. (2020). "Activation functions in neural networks". In: *International Journal of Engineering Applied Sciences and Technology (IJEAST* 4.12, pp. 310–316.

Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., and Sarro, F. (2024). "A survey on machine learning techniques applied to source code". In: *Journal of Systems and Software* 209, p. 111934.

Shaw, P., Uszkoreit, J., and Vaswani, A. (2018). "Self-attention with relative position representations". In: *arXiv preprint arXiv:1803.02155*.

Shen, Y., Dasgupta, S., and Navlakha, S. (2023). "Reducing Catastrophic Forgetting With Associative Learning: A Lesson From Fruit Flies". In: *Neural Computation* 35.11, pp. 1797–1819.

Shin, J. and Nam, J. (2021). "A survey of automatic code generation from natural language". In: *Journal of Information Processing Systems* 17.3, pp. 537–555.

Shrestha, N., Barik, T., and Parnin, C. (2021). "Unravel: A fluent code explorer for data wrangling". In: *The 34th Annual ACM Symposium on User Interface Software and Technology*, pp. 198–207.

Souza Baulé, D. de, Wangenheim, C. G. von, Wangenheim, A. von, and Hauck, J. C. (2020). "Recent Progress in Automated Code Generation from GUI Images Using Machine Learning Techniques." In: *J. Univers. Comput. Sci.* 26.9, pp. 1095–1127.

Stanford (2015). *Context-Free Grammars. Mathematical Foundations of Computing Course.* URL: https://web.stanford.edu/class/archive/cs/cs103/cs103.1164/fall1516/lectures/20/Slides20.pdf. (Accessed: 01 February 2024).

Su, Y., Hassan Awadallah, A., Wang, M., and White, R. W. (2018). "Natural language interfaces with fine-grained user interaction: A case study on web APIs". In: *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pp. 855–864.

Sun, J., Liao, Q. V., Muller, M., Agarwal, M., Houde, S., Talamadupula, K., and Weisz, J. D. (2022). "Investigating explainability of generative AI for code through scenario-based design". In: *27th International Conference on Intelligent User Interfaces*, pp. 212–228.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems* 27.

Svozil, D., Kvasnicka, V., and Pospichal, J. (1997). "Introduction to multi-layer feed-forward neural networks". In: *Chemometrics and intelligent laboratory systems* 39.1, pp. 43–62.

Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. (2020). "Intellicode compose: Code generation using transformer". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443.

Svyatkovskiy, A., Zhao, Y., Fu, S., and Sundaresan, N. (2019). "Pythia: Ai-assisted code completion system". In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2727–2735.

Tabesh, P., Mousavidin, E., and Hasani, S. (2019). "Implementing big data strategies: A managerial perspective". In: *Business Horizons* 62.3, pp. 347–358.

Taylor, P. (2023). *Amount of data created, consumed, and stored 2010-2020, with forecasts to 2025*. URL: https://www.statista.com/statistics/871513/worldwide-data-created. (Accessed: 03 November 2023).

TheDataScientist (2023). *Why Do Most Data Science Learners Fail?* URL: https://thedatascientist.com/the-data-scientist-march-newsletter-why-do-most-data-science-learners-fail-heres-a-free-call-with-an-expert-to-help-guide-you/. (Accessed: 19 December 2023).

Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., et al. (2022). "Lamda: Language models for dialog applications". In: *arXiv preprint arXiv:2201.08239*.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023a). "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971*.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. (2023b). "Llama 2: Open foundation and fine-tuned chat models". In: *arXiv preprint arXiv:2307.09288*.

Tu, Z., Su, Z., and Devanbu, P. (2014). "On the Localness of Software". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. ISBN: 978-1-4503-3056-5. Hong Kong, China: ACM, pp. 269–280. DOI: 10.1145/2635868.2635875. URL: http://doi.acm.org/10.1145/2635868.2635875.

Vaithilingam, P., Zhang, T., and Glassman, E. L. (2022). "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models". In: *Chi conference on human factors in computing systems extended abstracts*, pp. 1–7.

Van Der Aalst, W. (2016). *Data science in action*. Springer.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). "Attention is all you need". In: *Advances in neural information processing systems* 30.

Vincent, P., Iijima, K., Leow, A., West, M., and Matvitskyy, O. (2022). *Gartner Magic Quadrant for Enterprise Low-Code Application Platforms*. URL: https://www.gartner.com/en/documents/4022825. (Accessed: 02 January 2024).

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). "Pointer Networks". In: *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc., pp. 2692–2700. URL: http://papers.nips.cc/paper/5866-pointer-networks.pdf.

Waibel, T. (2021). "Improving a tool for using Domain Specific Languages (NLDSL)". BA thesis. Heidelberg University, Germany.

Walz, P. and Neef, J. (2020). *DeepDSL*. Heidelberg University, Germany.

Wang, L., Xu, W., Lan, Y., Hu, Z., Lan, Y., Lee, R. K.-W., and Lim, E.-P. (2023a). "Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models". In: *arXiv preprint arXiv:2305.04091*.

Wang, P., Fan, E., and Wang, P. (2021a). "Comparative analysis of image classification algorithms based on traditional machine learning and deep learning". In: *Pattern Recognition Letters* 141, pp. 61–67.

Wang, S., Geng, M., Lin, B., Sun, Z., Wen, M., Liu, Y., Li, L., Bissyandé, T. F., and Mao, X. (2023b). "Natural Language to Code: How Far Are We?" In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 375–387.

Wang, X., Wang, Z., Liu, J., Chen, Y., Yuan, L., Peng, H., and Ji, H. (2023c). "MINT: Evaluating LLMs in Multi-turn Interaction with Tools and Language Feedback". In: *The Twelfth International Conference on Learning Representations*.

Wang, Y. and Li, H. (2021b). "Code completion by modeling flattened abstract syntax trees as graphs". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 16, pp. 14015–14023.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. (2021c). "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation". In: *arXiv preprint arXiv:2109.00859*.

Weber, P. (2020). "Enhancing IDE Support for DSL Programming". BA thesis. Heidelberg University, Germany.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. (2022). "Chain-of-thought prompting elicits reasoning in large language models". In: *Advances in Neural Information Processing Systems* 35, pp. 24824–24837.

Weisz, J. D., Muller, M., Ross, S. I., Martinez, F., Houde, S., Agarwal, M., Talamadupula, K., and Richards, J. T. (2022). "Better together? An evaluation of AI-supported code translation". In: *27th International Conference on Intelligent User Interfaces*, pp. 369–391.

Wells, R. (2023). *The Future Of Work: 5 High-Demand Jobs In 2023*. URL: https://www.forbes.com/sites/rachelwells/2023/09/18/the-future-of-work-5-high-demand-jobs-in-2023. (Accessed: 31 October 2023).

Weng, G. and Andrzejak, A. (2023). "Automatic Bug Fixing via Deliberate Problem Solving with Large Language Models". In: *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, pp. 34–36.

Weng, L. (2018). *Attention? Attention!* URL: https://lilianweng.github.io/posts/2018-06-24-attention/. (Accessed: 12 February 2024).

Wenz, O. (2019). "An IDE-supported DSL for Data Science". BA thesis. Heidelberg University, Germany.

Wöhrer, M. and Zdun, U. (2020). "Domain specific language for smart contract development". In: *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, pp. 1–9.

Wu, S. (2024). *Introducing Devin, the first AI software engineer*. URL: https://www.cognition-labs.com/introducing-devin. (Accessed: 30 March 2024).

Wu, Y., Rabe, M. N., Hutchins, D., and Szegedy, C. (2022). "Memorizing transformers". In: *arXiv preprint arXiv:2203.08913*.

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., and Liu, T. (2020). "On layer normalization in the transformer architecture". In: *International Conference on Machine Learning*. PMLR, pp. 10524–10533.

Xu, F. F., Vasilescu, B., and Neubig, G. (2022). "In-IDE code generation from natural language: Promise and challenges". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.2, pp. 1–47.

Xue, M., Andrzejak, A., and Leuther, M. (2023). "An interpretable error correction method for enhancing code-to-code translation". In: *The Twelfth International Conference on Learning Representations*.

Yang, J., Jin, H., Tang, R., Han, X., Feng, Q., Jiang, H., Yin, B., and Hu, X. (2023). "Harnessing the power of llms in practice: A survey on chatgpt and beyond". In: *arXiv preprint arXiv:2304.13712*.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. (2024). "Tree of thoughts: Deliberate problem solving with large language models". In: *Advances in Neural Information Processing Systems* 36.

Yin, P. and Neubig, G. (2018). "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation". In: *arXiv preprint arXiv:1810.02720*.

Zaman, A. U. (2023). *Pandas vs PySpark..!* URL: https://medium.com/geekculture/pandas-vs-pyspark-fe110c266e5c. (Accessed: 22 April 2024).

Zeng, P., Lin, G., Pan, L., Tai, Y., and Zhang, J. (2020). "Software vulnerability analysis and discovery using deep learning techniques: A survey". In: *IEEE Access* 8, pp. 197158–197172.

Zhang, T., Damerau, F., and Johnson, D. (2002). "Text chunking based on a generalization of Winnow." In: *Journal of Machine Learning Research* 2.4.

Zhao, R., Chen, H., Wang, W., Jiao, F., Do, X. L., Qin, C., Ding, B., Guo, X., Li, M., Li, X., et al. (2023). "Retrieving multimodal information for augmented generation: A survey". In: *arXiv preprint arXiv:2303.10868*.

Part VI

Appendix

Data Science-related Jobs

Appendix A

A Glimpse of The Past Decade

The term "Data Science" emerged in the early 1960s when scientists observed the advancement of mathematical statistics with the aid from computers (Press, 2013; Foote, 2021). Throughout the years, data science continued to evolve and became one of the fastest growing fields across every industry[1]. To supplement our motivation mentioned in Chapter 1, this appendix presents a quick overview of jobs related to data science over the last ten years. Besides, relevant obstacles that data scientists and practitioners face while accomplishing their tasks are also discussed.

A.1 The Rise of Data Science-related Jobs

> Data science-related jobs are considered among the most rapidly-growing worldwide.

In the book *Data Science in Action*, Wil Van Der Aalst identified 12 main ingredients of data science, from statistics and algorithms to data mining, machine learning, visualization, and ethics (Van Der Aalst, 2016). He also pointed out that it is difficult to combine all different skills needed in a single person, i.e. a *unicorn data scientist*. Therefore, it is unsurprising that data scientist was appraised as the *sexiest job*[2] in 2012 (Davenport et al., 2012).

Ten years later, data scientist retains its place in the list of high-demand occupations together with a variety of other data science-related jobs (Davenport et al., 2022; Wells, 2023; Gafner, 2023). A series of *Jobs on the Rise* reports published by LinkedIn unveils a consistent presence of data science positions in the past three years (Anderson, 2021; Hilgers, 2022; Hilgers, 2023). Additionally, a brief analysis of search interest in data science professions also augments this job trend. To illustrate the ascent of these jobs over the past decade, we compare them with a pivotal role, human resource, which remains essential irrespective of the pace of technological advancement (Ammirato et al., 2023).

Figure A.1 presents the Google Trends[3] of the four search terms *data scientist*, *data analyst*, *data engineer*[4], and *human resource* from 2013 to 2023. It is intriguing to observe that the trends of *data scientist*, *data analyst*, and *human resource* gradually converge over the years from 2013 to 2022 in the favor of the *human resource* search term. The notable shift occurred in July of

[1] What is data science, https://www.ibm.com/topics/data-science.
[2] The term is utilized by Thomas H. Davenport and DJ Patil to emphasize the high demand of the job.
[3] Google Trends, https://trends.google.com/trends/explore.
[4] Job titles are obtained from a study program of University of Adelaide, Australia, https://online.adelaide.edu.au/blog/data-scientist-vs-data-analyst-vs-data-engineer. (Accessed: 03 November 2023).

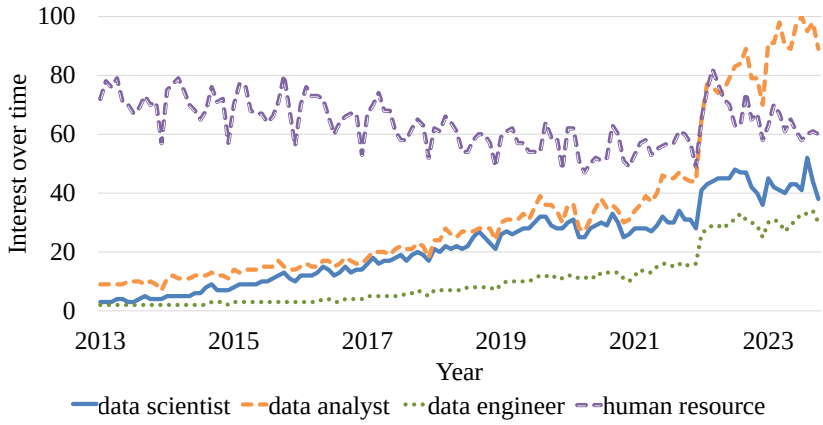


Figure A.1: Interest over time* of data science-related jobs from 2013 to 2023 via Google Trends.

* The numbers exhibit search interest relative to the highest point on the chart for the given region and time.

2023 when the term *data analyst* reached the peak popularity of the period and surpassed all the remaining keywords. This finding reaffirms that data science has achieved extensive recognition.

Nevertheless, the accelerated development of data science has unveiled not only a myriad of opportunities but also unneglectable challenges. Multiple surveys have been conducted to gain deeper insights into this fast-growing discipline. In the subsequent sections, we summarize the main obstacles which foster the inspiration for this dissertation.

A.2 Time Allocation for a Data Scientist

> Data scientists generally allocate a significant portion of their time to data cleansing and preparation, a practice that remains accurate until recently.

CrowdFlower published their first *Data Scientist Report* in 2015, pointing out that cleaning and organizing data is the most time-consuming task and also the least interesting part of data scientists' jobs, cited by two-thirds of their survey respondents (CrowdFlower, 2015). To further clarify this issue in recent years, Figure A.2 epitomizes reports published from various sources in the 2016–2022 period for the question "How do data scientists spend their time?".

Overall, data scientists are dedicating reduced amounts of time on data cleaning and preparation over the years. However, these tasks still account for over one-third of their time as estimated in the 2022 survey (Anaconda, 2022). Specifically, until 2016, data scientists devoted nearly 80% of their time for preprocessing and preparing data, namely 60% for cleaning and organizing data and 19% for collecting data sets (CrowdFlower, 2016).

These portions in total were then significantly decreased to more than a half (i.e. 51%) as reported in CrowdFlower (2017). While the surveys published in 2018 left out the considering question (FigureEight, 2018; Anaconda, 2018), a report distributed by Appen (2019)[5] addressed a similar inquiry, which reveals 73.5% of their technical respondents spend 25% or more of their time on managing, cleaning, and/or labeling data. Ultimately, statistics disclosed in 2020–2022

[5] CrowdFlower is the former name for both FigureEight and Appen.

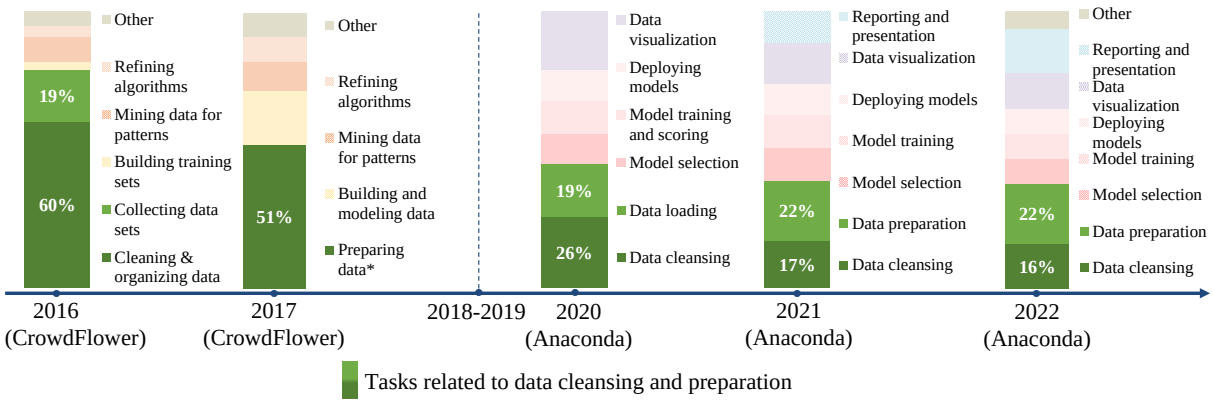


Figure A.2: Summary of surveys from multiple sources from 2016 to 2022** on the question "How do data scientists spend their time?".

* The authors grouped collecting, labeling, cleaning and organizing data into one task.
** Surveys published in 2018 and 2019 do not cover the exact considering question.

present a slight decline of time consumed for tasks related to data cleaning and preparation, from 45% to 39% and 38%, respectively (Anaconda, 2020; Anaconda, 2021; Anaconda, 2022).

By 2023, although surveys released in this year did not cover the exact same questionnaire, participants still reaffirmed the trend. Namely, data preparation and data cleaning are the two most time consuming tasks (Anaconda, 2023), and also the second most regular and challenging steps in the data science lifecycle, with data access being the first one (Knime, 2023).

In other words, lessening time for preparing data would boost performance of data scientists in their more interesting tasks, e.g. mining data for patterns, building and modeling data, and refining algorithms (CrowdFlower, 2017). However, surveys over the past decade also uncovered numerous programming languages and data science tool-kits that impede data scientists in efficiently accomplishing their tasks (BARC, 2015; Knime, 2023). The following section briefly discusses these typical obstacles.

A.3 Technical and Analytical Know-how Problem

> Researchers and practitioners must engage a plethora of programming languages and data science ⚒ tool-kits.

⚒ Microsoft Excel was formerly reported as the most commonly used tool among data scientists, along with 47 other programming languages and tools, such as Python, R, ⚒ Tableau[6], and ⚒ SAS[7] (CrowdFlower, 2015). However, from 2016 to 2023, Python has gradually overtaken this position, becoming the most popular programming language of data science (Databricks, 2023). Notably, a report of Anaconda (2018) illustrates that Matplotlib (i.e. a plotting library for Python) was the most used visualization tool cited by 74.82% of respondents, while Excel

[6]Business intelligence and analytics software, https://www.tableau.com/.
[7]Statistical Analysis Software, https://www.sas.com/en_us/home.html.
(Accessed: 22 November, 2023)

belongs to the *Other* category (9.39%), alongside ⚒ Microsoft Power BI[8], ⚒ TIBCO Spotfire[9], ⚒ Qlik[10], and ⚒ Altair[11].

Additionally, surveys released since 2018 observed a proliferation of Machine Learning (ML) frameworks, including Pandas, NumPy, Scikit-learn, and around 25 other frameworks, such as TensorFlow, PyTorch, and Seaborn (FigureEight, 2018; Appen, 2019). Recent reports on data science from Anaconda (2020; 2021; 2022) consider six to 14 popular programming languages used by data scientists, researchers, students, and professionals.

These numbers reiterate the rapid evolution of the data science field, where new tools and techniques continuously emerge. Data scientists and practitioners must keep pace with the latest developments and acquire new skills for effective task completion (TheDataScientist, 2023).

> Practitioners face the roadblock of switching between programming languages when moving their data science models to production environments.

In 2020, Anaconda highlighted that the competitive advantage in data science stems from deploying ML models and other outputs. However, transitioning to production poses challenges beyond the control of data professionals (Anaconda, 2020). Notably, 17% of data scientists, 27% of developers, and 11% of system administrators faced the roadblock of re-coding models from Python/R to another language.

The surveys published by Anaconda two years later unveiled similar trends, with the ratio reaching 26% of total respondents in 2022 (Anaconda, 2021; Anaconda, 2022). Additionally, re-coding models from another language to Python/R is also added to their top-10 obstacles. In other words, this roadblock results in the need of facilitating translation from one programming language to another with minimum effort for end-users.

[8] Data visualization software, https://www.microsoft.com/en-us/power-platform/products/power-bi.
[9] Analytics platform, https://www.spotfire.com/.
[10] Business analytics platform, https://www.qlik.com/us/.
[11] Software and cloud solutions for simulation, Internet of Things, high performance computing, data analytics, and artificial intelligence, https://altair.com/company.
(Accessed: 22 November, 2023).

Artificial Neural Network

Appendix B

Fundamental Concepts and Techniques

This appendix provides fundamental concepts and techniques of neural networks, establishing the foundation for comprehending the underlying architectures of models discussed in Chapter 2, Section 2.2.4. For further explanation of neural networks and related techniques, we refer to Deep Learning (DL) books such as Bengio et al. (2017), Kelleher (2019) or videos from 3Blue1Brown (2017) for visual effects.

B.1 Overview of Artificial Neural Networks

Neural networks or Artificial Neural Networks (ANNs) encompass processing elements known as *neurons* or *nodes*, arranged in layers. A standard ANN comprises one *input layer*, one or more *hidden layers* (at least two in DL networks), and one *output layer*, mimicking the structure of the human brain (Svozil et al., 1997). Figure B.1 illustrates an ANN architecture with two hidden layers.

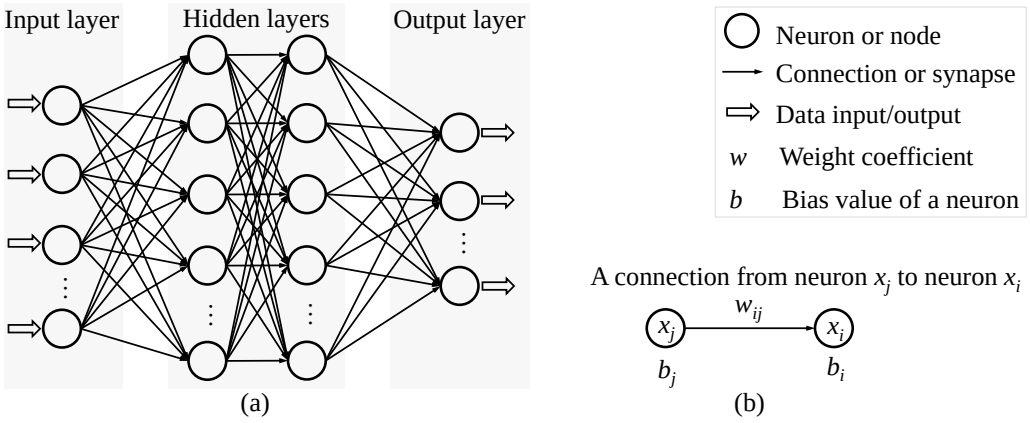


Figure B.1: An example of ANN with two hidden layers.

Typically, each neuron in the network receives input signals, processes them, and emits an output signal. Neurons in one layer are linked to at least one neuron in the next layer via *connections*, also known as *synapses*. Each connection is assigned a *weight coefficient*, a real number representing its significance in the network (Svozil et al., 1997).

The output signal of a neuron, known as its *activation value*, indicates the activated level of the neuron in influencing the network's output. The neuron convert input signals (e.g. numeric values) into an activation level by multiplying each input value by a weight w, summing the

results, and then passing the weighted sum plus the neuron's bias value b through an *activation function* f. Considering neuron x_i and all neurons x_j connected to it in Figure B.1(b), the output value of x_i is defined by Equations B.1 and B.2.

$$x_i = f(z_i) \tag{B.1}$$

$$z_i = (\sum_j w_{ij} * x_j) + b_i \tag{B.2}$$

It is worth mentioning that most activation functions in ANNs are nonlinear, facilitating the learning of nonlinear mappings between inputs and outputs, capturing complex relationships inherent in real-world data. Commonly utilized functions include `sigmoid`, `tanh`, `ReLU`, and `softmax` (Sharma et al., 2020). Additionally, the progress of learning or training a neural network involves weight coefficient adjustments to fulfill specific conditions, such as minimizing output deviation from target output in supervised learning, or attaining the desired number of data clusters in unsupervised learning (Svozil et al., 1997; Kelleher, 2019).

B.2 Feed-forward and Recurrent Neural Network

Feed-forward Neural Network (FFNN) is one of the most popular and classical neural networks, which does not allow feedback between layers and transmits information in one direction only, hence the name *feed-forward* (Svozil et al., 1997). Besides, FFNNs are fully connected, with each neuron receiving all activation values from all neurons in the preceding layer (Abiodun et al., 2018). FFNNs are employed in multiple tasks such as classification and clustering. The ANN displayed in Figure B.1 is also a FFNN.

Recurrent Neural Network (RNN) is another prominent ANN architecture, designed for sequential data processing. RNN is a feedback neural network with an input layer, a context layer or a hidden layer with a memory buffer, and an output layer. The hidden layer of an RNN extends over time, with connections feeding into the subsequent time steps rather than the next layer as in FFNN (Kelleher, 2019). Figure B.2 illustrates the flow of information in an RNN through time steps, given that `seqlen` is the length of the input sequence.

At time step t, each neuron in the hidden layer receives input from the input layer and the memory buffer. Activation values generated by these neurons are forwarded to the output layer and concurrently stored in the memory buffer, overwriting the previous values (time step $t+0.5$). At the subsequent time step ($t+1$), a new input from the sequence is fetched, restarting the whole process. The memory buffer solely retains information calculated by the hidden layer without processing it. In other words, there are no weight coefficients on connections from hidden layer to memory buffer. However, weight coefficients still exist on other network connections.

With this architecture, the depth of an RNN correlates with the length of the input sequence (Kelleher, 2019). Figure B.3 shows the unrolled representation of the RNN architecture through time, where h_0 denotes the initial state of the memory buffer, and $h_1, h_2, ..., h_{seqlen}$ indicate the hidden layer at each time step t.

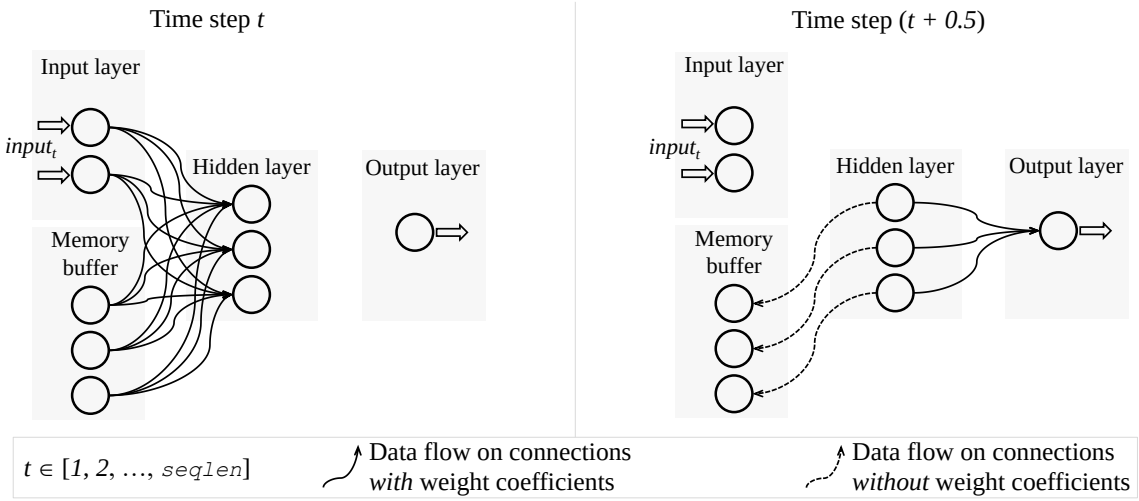


Figure B.2: The flow of information in an RNN handling a sequence of inputs with length `seqlen`, adapted from Figure 5.2 of Kelleher (2019).

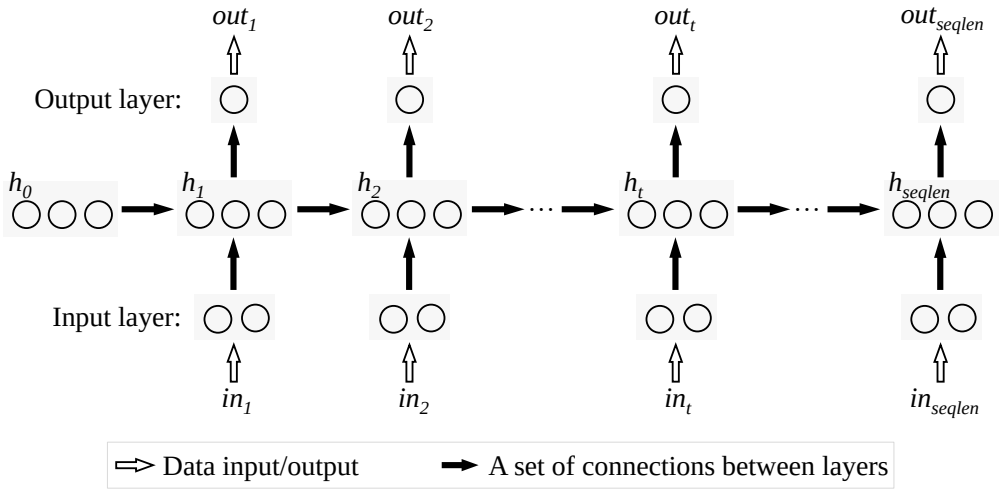


Figure B.3: Unrolling the RNN architecture depicted in Figure B.2 through time, adapted from Figure 5.3 of Kelleher (2019).

B.3 Problems with Recurrent Neural Networks

RNNs encounter two widely-known issues when training the neural networks, namely *vanishing* and *exploding* gradient.

Vanishing gradient problem. Despite the capability in processing sequential data, RNNs face the problem of vanishing gradients, hindering their effectiveness with long sequences. This challenge arises from RNNs being trained using the backpropagation algorithm (Svozil et al., 1997), feeding backward the error, i.e. the disparity between generated output and target output, across the entire network to adjust weight coefficients.

This backpropagation involves iteratively multiplying the error by the same set of weight coefficients through all states of the hidden layer (i.e. weights from the memory buffer to the hidden layer). Consequently, if these weights are less than 1, the error gradient diminishes exponentially concerning the input sequence length, leading to its vanishment. Additionally,

updates to weight coefficients in the early states (e.g. $t = 1$) of the hidden layer are insignificant, thus prolonging model training process (Kelleher, 2019).

Exploding gradient problem. Exploding gradients arise from initial weight coefficients exceeding 1, leading to amplified values during forward or backpropagation. In RNNs, large weights can cause unstable networks or chaos, where activation levels in forward propagation are extremely sensitive to small perturbations of the input (Equation B.2).

Furthermore, excessive weight coefficients may drive activation functions towards saturation (e.g. reaching 0 or 1 with the `sigmoid` function), resulting in gradient loss through saturated units. A possible solution to alleviate the exploding gradient problem is gradient clipping, where gradient values are bounded by thresholds prior to a gradient descent step (Bengio et al., 2017).

B.4 Long Short-Term Memory

Long Short-Term Memory (LSTM), introduced by Hochreiter et al. (1997), enhances RNNs by mitigating the vanishing gradient problem through the elimination of repetitive multiplication of same weight coefficients. Each core processing unit of LSTM, named *cell*, comprises three *gates*: (i) *forget gate*, (ii) *input gate*, and (iii) *output gate*. These gates regulate the flow of activation values within the cell over time. Each gate consists of layers of standard neurons, with one neuron per activation value in the cell state (Kelleher, 2019). Figure B.4 demonstrates the flow of information within an LSTM cell, with explanations of data parameters provided in Table B.1. The input and output layers are excluded in the diagram.

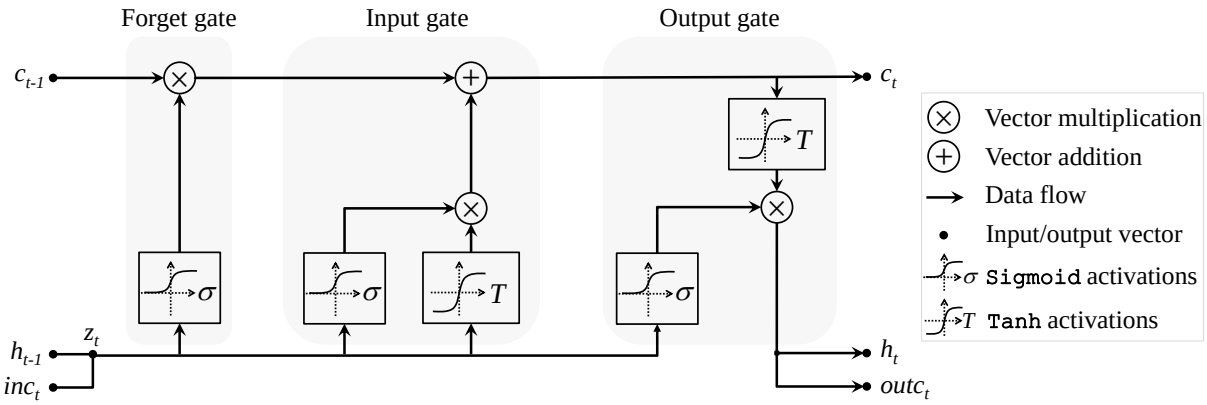


Figure B.4: The flow of information within an LSTM cell at time step t, adapted from Figure 5.4 of Kelleher (2019). The input and output layers are omitted.

At time step t, input of the cell inc_t from the input layer and the preceding hidden state h_{t-1} are concatenated to create vector z_t, which is then injected into each gate of the LSTM cell. The forget gate determines which elements of the cell state c_{t-1} should be forgotten at time step t, based on the input z_t. This operation involves element-wise multiplication between c_{t-1} and the `sigmoid` vector $\sigma(z_t)$. Elements of $\sigma(z_t)$ approaching 0 indicate corresponding values in c_{t-1} to be forgotten, whereas values close to 1 indicate retention.

Subsequently, the input gate controls the updating of activation values within the cell at time step t by firstly utilizing the `sigmoid` vector $\sigma(z_t)$ to select which elements of c_{t-1} should be

Table B.1: Meaning of data parameters in Figure B.4.

Parameter	Meaning
c_t, c_{t-1}	vector (of activation values) of the cell state at time step t and the preceding time step
h_t, h_{t-1}	vector of hidden layer state at time step t and the preceding time step
inc_t, $outc_t$	vectors of input and output values of the cell at time step t
z_t	concatenated vector from inc_t and h_{t-1}
$\sigma(z_t)$	`sigmoid` vector with values in range $(0, 1)$
$T(z_t)$	`tanh` vector with values in range $(-1, 1)$

updated, similarly to the forget gate. Secondly, the `tanh` vector $T(z_t)$ is employed to calculate the update values to be incorporated into the cell state. The product of vectors $\sigma(z_t)$ and $T(z_t)$ denotes the update values for specific elements in c_{t-1}. Consequently, the activation values of the cell state at time step t can be adjusted, i.e., $c_t = c_{t-1} + (\sigma(z_t) \times T(z_t))$.

Ultimately, the output gate decides which activation values in the cell state c_t contribute to the cell's output value, denoted as $outc_t$. The cell state c_t undergoes processing through a `tanh` layer (composed of neurons utilizing the `tanh` activation function) to derive candidate values. Subsequently, the `tanh` vector $T(z_t)$ is multiplied by the `sigmoid` vector $\sigma(z_t)$ to filter relevant elements. The resultant vector ($outc_t$) is propagated to the output layer, while also becoming the hidden layer state h_t of time step t (Kelleher, 2019).

Here, the structure of LSTM alleviates the vanishing gradient problem by refraining from exponentially fast decaying factors in the error gradient after applying the backpropagation algorithm (Bayer, 2015). For simplicity, relevant equations are omitted in our discussion.

LSTM architecture is particularly suitable for NLP (Kelleher, 2019). This neural network is deployed in various code completion models, including those encoding Abstract Syntax Tree (AST) paths (Svyatkovskiy et al., 2019; Alon et al., 2020b) or in combination with attention mechanism (Li et al., 2018; Rahman et al., 2020).

B.5 Word Embedding

To enable neural networks for NL or source code processing, transformation of words or code tokens into numerical vectors is essential. Tomas Mikolov proposed notable methods for this conversion. Namely, instead of utilizing *n-gram* models, which lack inherent relationships to one another, or sparse vectors obtained by one-hot encoding, Mikolov et al. (2013c) investigated a dense vector representation for words that preserves both syntactic and semantic properties. These vectors are so called word embeddings.

The underlying concept is that words that have analogous context, i.e. surrounding words, would have similar meanings (Kelleher, 2019). Consequently, such words exhibit proximity in vector space, as determined by vector offset, often measured via cosine distance. For instance, the operation `vector(King) - vector(Man) + vector(Woman)` yields a vector closely resembling `vector(Queen)`, exemplifying this phenomenon.

Mikolov et al. (2013c) employed an RNN architecture to derive word vector representations, implicitly learned by the input layer weights. Each word is encoded using 1-of-N coding (i.e. one-hot encoding), with the output layer generating a probability distribution across words. The dimensionality of both input and output vectors corresponds to the vocabulary size.

In their subsequent publication, Mikolov et al. (2013a) introduced two models for Word2Vec transformation, drawing inspiration from a probabilistic FFNN language model (Bengio et al., 2000): Continuous Bag-of-Words (CBOW) and Continuous Skip-gram. While CBOW learns the representation of a word by predicting the word, given its context, Continuous Skip-gram gains the representation by predicting the context, given the word itself.

B.6 Variants of the Attention Mechanism

Originally proposed by Bahdanau et al. (2015), the *attention mechanism* aims to address the *hidden state bottleneck* problem (i.e. all information is compressed into a fixed-size vector) in Seq2seq models (details in Section 2.2.4 of Chapter 2). Luong et al. (2015) simplified and generalized the attention architecture under several aspects, as summarized in Table B.2.

Table B.2: Variants of attention architectures in Bahdanau et al. (2015) and Luong et al. (2015).

Aspect	**By Bahdanau et al. (2015)**	**By Luong et al. (2015)**
Architecture	bidirectional encoder with unidirectional decoder	stacking LSTM for both encoder and decoder
Computation path	$d_{t-1} \rightarrow \alpha_i^t \rightarrow c_t \rightarrow d_t \rightarrow out_t$	$d_t \rightarrow \alpha_i^t \rightarrow c_t \rightarrow atn_t \rightarrow out_t$ where $atn_t = \tanh(W_C[c_t; d_t])$ is the attention hidden state
Decoder hidden state d_t	computed by a gated hidden unit from d_{t-1}, out_{t-1}, and c_t	obtained from the decoder at time step t
Attention score	*additive*: $v_A^\top \tanh(W_A d_{t-1} + U_A e_i)$	*concat*[*]: $v_A^\top \tanh(W_A[d_t; e_i])$
Output word out_t	using deep output with a maxout hidden layer on d_t, out_{t-1}, and c_t	applying `softmax` on atn_t

[*] This score function is utilized for global attention. Besides *concat*, Luong et al. (2015) considered two alternatives, including *dot-product* and *general*.
W_C is a learnable parameter, other symbols are explained in Section 2.2.4 of Chapter 2.

For instance, to calculate the output word out_t, Luong et al. (2015) simply concatenated the decoder hidden state d_t and the context vector c_t to create an attention hidden state atn_t. The `softmax` value of atn_t determines the output word at time step t. In addition, Luong et al. (2015) used stacking LSTM, instead of bidirectional RNN, for the encoder and an unidirectional RNN for the decoder, in contrast to Bahdanau et al. (2015).

Furthermore, Luong et al. (2015) categorized their approach into global attention, which attends to all source words, and local attention, which considers only a subset of source words at a time. The former (global attention) aligns with the concept proposed by Bahdanau et al. (2015), as all encoder hidden states are used to derive the context vector. Consequently, the context vector varies in length, matching the number of time steps to process the input sequence.

In the latter (local attention), Luong et al. (2015) determine an aligned position p_t for each output word at time step t. The context vector is computed as a weighted average over the set of encoder hidden states within the window $[p_t - D, p_t + D]$, where D is empirically determined. While these two attention types differ in how the context vector c_t is constructed, they share subsequent steps, involving transforming c_t into the attention hidden state atn_t and predicting the output word out_t, as outlined in Table B.2.

The end of this manuscript is reached here.
We hope that our work brought you useful insights.
Thank you for your time.