Code Completion
CMPE 540 Final Project

Damla Şentürk
2022700252

January 17, 2024

1 Introduction

1.1 Background

In the realm of software development, efficiency and accuracy are paramount. As the complexity of software systems escalates, developers increasingly rely on tools that assist in writing code. One such tool is code completion, a feature commonly found in Integrated Development Environments (IDEs) that predicts and suggests the next segment of code a developer might write. This not only accelerates the coding process but also reduces the likelihood of syntactical errors, thereby improving code quality.

The importance of this problem extends beyond individual convenience. In a broader context, efficient code completion tools can significantly reduce development time, leading to faster project turnarounds and more time for developers to focus on complex problem-solving and innovation. They also have the potential to lower the entry barrier for novice programmers, making coding more accessible and less intimidating. Furthermore, in an age where software is omnipresent, enhancing the tools that build these software can have a multiplier effect on productivity and quality across various industries.

1.2 Problem Statement

Despite its utility, the current state of code completion technologies is not without shortcomings. Traditional code completion tools often rely on relatively simplistic algorithms that suggest code based on lexical matching or statistical models trained on large codebases. While effective to a degree, these methods do not fully capture the nuanced and context-specific nature of programming. This leads to suggestions that are syntactically correct but semantically inappropriate or irrelevant to the current coding context.

1.3 Motivation

This project is motivated by the need for a more sophisticated approach to code completion, one that understands the intricacies of the developer's intent and the specific constraints of the coding environment. By framing code completion as a Constraint

Satisfaction Problem (CSP), we aim to create a system that intelligently navigates the complexities of programming languages and offers contextually relevant code suggestions.

1.4 Objectives

The primary objectives of this project are as follows:

- **To Develop a CSP-Based Code Completion Model:** We aim to design and implement a model that uses the principles of constraint satisfaction to predict and suggest code snippets.
- **To Evaluate the Model's Effectiveness:** Through rigorous testing and comparison with traditional code completion methods, we intend to assess the practicality and accuracy of our CSP-based model.
- **To Explore the Model's Scalability and Adaptability:** An investigation into how well the model adapts to different programming languages and coding scenarios is essential for understanding its potential for widespread adoption.

1.5 Scope of the Project

This project focuses on developing a model for code completion using constraint satisfaction techniques. The scope includes designing the model, implementing a proof of concept, and evaluating its performance with selected programming languages. While the project aims to lay a foundational framework for CSP-based code completion, it does not seek to create a market-ready product but rather to explore the feasibility and potential of this approach.

1.6 Structure of the Report

The report is structured as follows: Chapter 2 presents a literature review, highlighting existing code completion methodologies and their relation to our approach. Chapter 3 describes the dataset and preprocessing steps; detail the baseline, oracles, and our main approach; and then discusses the evaluation metrics, followed by results and analysis in Chapter 4. The report concludes with future work in Chapter 5.

2 Related Work

Understanding the landscape of existing research and developments in code completion is crucial for contextualizing the Constraint satisfaction-based code completion system. This chapter reviews relevant literature and projects, highlighting their methodologies, achievements, and how they relate to or differ from our work.

2.1 Traditional Code Completion Techniques

Code completion is a crucial feature in modern software development tools that aims to assist developers by automatically suggesting code snippets or completing code statements

based on the context. Over the years, researchers have explored various approaches and techniques to improve code completion accuracy and efficiency.

One of the early works in code completion is the Probabilistic Model for Code with Decision Trees proposed by Raychev et al. (2016). They introduced a probabilistic model that leverages decision trees to predict the next code token given the previous tokens. The model achieved promising results in predicting code tokens, but it had limitations in handling complex code structures and dependencies. Future research could focus on enhancing the model's ability to capture higher-level code semantics and improve its performance on large-scale codebases.

Another approach to code completion is the use of neural networks and attention mechanisms. Li et al. (2017) proposed a Code Completion model with Neural Attention and Pointer Networks. Their model utilized attention mechanisms to capture the relevance of different code tokens and employed pointer networks to handle out-of-vocabulary tokens. The model demonstrated improved code completion accuracy compared to traditional methods. However, the model's performance could be further enhanced by incorporating more advanced neural network architectures and exploring different attention mechanisms.

In recent years, transformer-based models have gained popularity in various natural language processing tasks. Svyatkovskiy et al. (2020) introduced IntelliCode Compose, a code generation model based on transformers. Their model leveraged the transformer architecture to generate code snippets based on natural language descriptions. The model showed promising results in generating code, but it had limitations in handling complex code logic and ensuring syntactic correctness. Future research could focus on refining the transformer-based models for code completion by incorporating domain-specific knowledge and integrating syntactic and semantic constraints.

Furthermore, the research findings by Kats and Visser (2010) introduced the Spoofax Language Workbench, which provides rules for declarative specification of languages and IDEs. This work contributes to the development of tools and frameworks that facilitate code completion by enabling the specification of language-specific rules and constraints. However, there is a need for further research to explore the scalability and applicability of such language workbenches in real-world software development scenarios.

3 Methodology

This section outlines the methodologies employed in the project, detailing the foundation upon which our research is built, detailing the robust dataset meticulously curated for training, the baseline and oracle models established for comparative analysis, the main approach devised leveraging CSP, and the multifaceted evaluation metrics adopted to rigorously measure the success and impact of our project.

3.1 Dataset

A crucial element in developing a constraint satisfaction-based code completion system is the selection and preparation of an appropriate dataset. The dataset not only forms the foundation upon which the model is trained and tested but also defines the scope and applicability of the system. For this project, we have selected a diverse and comprehensive

code dataset that encompasses a range of programming scenarios.

3.1.1 Data Collection

The dataset chosen for this project, CodeSearchNet by Husain et al. (2019) hosted on HuggingFace, is derived from open-source repositories, primarily GitHub. It includes a variety of programming languages, but for the initial phase, we focus on Python due to its widespread usage and clear syntax. The dataset comprises a mix of code snippets, ranging from simple function definitions to more complex class implementations and algorithmic solutions.

3.1.2 Preprocessing

The preprocessing steps for preparing the dataset are as follows:

- **Tokenization:** Each line of code is broken down into tokens (keywords, operators, variables, etc.).
- **Syntactic Analysis:** Parsing the code to understand its syntactic structure, which aids in identifying constraints.
- **Normalization:** Standardizing variable names and formatting to maintain consistency across the dataset.
- **Annotation:** Manual annotation of code segments with metadata that describes their context and purpose, aiding the CSP model in understanding coding patterns.

3.2 Baseline and Oracles

In the development of the constraint satisfaction-based code completion system, establishing baselines and oracles is crucial. Baselines provide a point of reference to measure the effectiveness of our model, while oracles offer an idealistic performance target. This chapter describes the chosen baselines and oracles for our project.

3.2.1 Baselines

The purpose of baselines is to provide a simple yet effective comparison against more complex models like ours. For this project, we employ one primary baseline: Simple Rule-Based System. This system suggests code completions based on a predefined set of rules derived from common coding patterns. For instance, if a user types if, the system might automatically suggest if condition: as a completion.

The rules used for the rule-based system are as follows:

- Keyword Completion:
  - If a line starts with a common keyword (e.g., def, for, if, $class$), automatically suggest the typical structure that follows (e.g., $def function_name()$:).
  - Suggest $elif$ or $else$ after an if or $elif$ statement.

```
{'code': 'def get_vid_from_url(url):\n'
         '        """Extracts video ID from URL.\n'
         '        """\n'
         "        return match1(url, r'youtu\\.be/([^?/]+)') or \\\n"
         "          match1(url, r'youtube\\.com/embed/([^/?]+)') or \\\n"
         "          match1(url, r'youtube\\.com/v/([^/?]+)') or \\\n"
         "          match1(url, r'youtube\\.com/watch/([^/?]+)') or \\\n"
         "          parse_query_param(url, 'v') or \\\n"
         "          parse_query_param(parse_query_param(url, 'u'), 'v')",
 'code_tokens': ['def',
                 'get_vid_from_url',
                 '(',
                 'url',
                 ')',
                 ':',
                 'return',
                 'match1',
                 '(',
                 'url',
                 ',',
                 "r'youtu\\.be/([^?/]+)'",
                 ')',
                 'or',
                 'match1',
                 '(',
                 'url',
                 ',',
                 "r'youtube\\.com/embed/([^/?]+)'",
                 ')',
                 'or',
                 'match1',
                 '(',
                 'url',
                 ',',
                 "r'youtube\\.com/v/([^/?]+)'",
                 ')',
                 'or',
                 'match1',
                 '(',
                 'url',
                 ',',
                 "r'youtube\\.com/watch/([^/?]+)'",
                 ')',
                 'or',
                 'parse_query_param',
                 '(',
                 'url',
                 ',',
                 "'v'",
                 ')',
                 'or',
                 'parse_query_param',
                 '(',
                 'parse_query_param',
                 '(',
                 'url',
                 ',',
                 "'u'",
                 ')',
                 ',',
                 "'v'",
                 ')'],
 'docstring': 'Extracts video ID from URL.',
 'docstring_tokens': ['Extracts', 'video', 'ID', 'from', 'URL', '.'],
```

Figure 1: Example data from the CodeSearchNet dataset.

- Indentation Management:
  - Automatically indent the next line after a colon : indicating a block start (e.g., after $if condition$:, $for i in range()$:, $class MyClass$:).
  - Dedent when likely ending a block (e.g., after *return*, *break*, *continue*).
- Function Snippets:
  - When def is typed, suggest a basic function template like $def function_name()$:.
  - Provide common function names as suggestions based on the context (e.g., $calculate_sum$, $retrieve_data$).
- Variable and Type Suggestions:
  - Suggest common variable names (e.g., *count*, *name*) after typical type declarations.
  - Offer built-in methods based on detected variable types (e.g., suggesting $.append()$ after a list variable).
- Control Structures:
  - Suggest $elif$ condition: or $else$: after an if statement block.
  - Offer for loop templates, like $for i in range()$: or $for item in collection$:.
  - Suggest try :, followed by $except Exception Name$: for error handling.
- Comment and Documentation Templates:
  - When """ is typed at the beginning of a function or class, suggest a standard docstring template.
  - Auto-complete common comment structures like $\#TODO$: or $\#FIXME$:.
- Error Handling Suggestions:
  - Suggest try : block followed by $except Exception$: when potential errors are detected.
  - Offer standard exceptions to catch (e.g., $ValueError$, $TypeError$, $KeyError$).
- Import Statements:
  - Suggest common library import statements based on the detected functions/variables (e.g., $import numpy as np$ if arrays are used).
- List and Dictionary Comprehensions:
  - Suggest list or dictionary comprehension structures when loops or mapping patterns are detected.
- Contextual Predictions:
  - Based on the most recent tokens or lines of code, suggest the most likely next token. For example, after $print($, suggest variables or strings commonly used.

3.2.2 Oracles

Oracles represent an idealised performance level that may not be practically achievable but offer a target to aspire to. For this project, the following oracles are considered:

- **Human Performance:** Members of the project team manually complete given code snippets. This performance level, although not perfect, represents a high standard of contextual understanding and logical reasoning.
- **Training Error of an Expressive Classifier:** Using a highly expressive classifier (like a large language model trained on code completion) and measuring its error on the training set serves as an oracle. This gives us an insight into the best possible performance under the constraint of the available dataset.

The selection of baselines and oracles was guided by their relevance and practicality in the context of code completion. The chosen baselines offer a spectrum of complexity, from rule-based to statistical, while the oracles provide a mix of human intuition and algorithmic perfection.

3.3 Main Approach

The aim is to formulate code completion as a constraint satisfaction problem (CSP), where the constraints are derived from the coding context, and the solutions are viable code completions.

3.3.1 Theoretical Foundation

The CSP framework operates by identifying constraints in a given coding scenario and then finding solutions that satisfy these constraints. In the context of code completion, constraints include syntactic rules of the programming language, semantic context from the surrounding code, and best coding practices.

3.3.2 Components of the Proposed Solution

The proposed solution comprises several key components:

- **Constraint Extractor:** This module analyzes the coding environment to identify constraints. It uses parsers and custom algorithms to understand variable types, expected function arguments, control flow structures, and other context-specific information.
- **Solution Generator:** Based on the identified constraints, this module generates possible code completions. It employs a combination of rule-based logic and probabilistic models to propose solutions that not only satisfy the constraints but also align with common coding patterns.
- **Evaluation Engine:** Once potential solutions are generated, this module ranks them based on various criteria such as the likelihood of correctness, adherence to coding standards, and contextual relevance.

3.3.3 Model and Algorithm

The proposed model operates in the following steps:

- **Input Processing:** The current partial code snippet is taken as input.
- **Constraint Extraction:** The Constraint Extractor parses the input to identify coding constraints.
- **Solution Generation:** The Solution Generator uses these constraints to create a list of possible code completions.
- **Solution Evaluation:** The Evaluation Engine ranks these completions, and the highest-ranked completion is suggested to the developer.

To illustrate, consider a scenario where the developer has typed $defcalculate($. The proposed model would work as follows:

1. The input is $defcalculate($.
2. The constraint extraction algorithm identifies the context as a function definition.
3. The solution generation algorithm generates potential parameters and types based on similar functions in the dataset.
4. Completions are ranked, and a suggestion like $defcalculate(value1, value2)$: is made.

Some challenges in developing the proposed method include handling ambiguous coding contexts and managing the computational complexity. Solutions involve enhancing the model with machine learning to better understand context and optimizing algorithms for efficiency.

3.4 Evaluation Metrics

To assess the effectiveness of the Constraint Satisfaction-Based Code Completion, it is essential to establish a set of robust evaluation metrics. These metrics will not only quantify the performance of the model but also allow for a meaningful comparison with the baselines and oracles. This chapter outlines the key metrics used in evaluating the proposed model.

The choice of metrics is driven by the need to capture various aspects of code completion effectiveness, including accuracy, relevance, and efficiency. The following metrics have been selected:

- **Accuracy:** Measures the proportion of suggestions made by the system that are correct.
- **Precision:** Assesses the relevance of the suggestions made by the model.
- **Recall:** Evaluates the system's ability to provide all relevant suggestions.

- **F1 Score:** Provides a balance between precision and recall, especially useful in scenarios where there is a trade-off between these two metrics.
- **Response Time:** Measures the time taken by the system to generate a suggestion, crucial for real-time code completion applications.

The evaluation of the proposed method using these metrics provides a comprehensive picture of its performance. It enables the identification of strengths and areas for improvement, guiding future development efforts.

4 Experimentation and Results

This chapter presents the results obtained from the implementation of the Constraint Satisfaction-Based Code Completion and analyses its performance in comparison to the established baselines. The analysis aims to provide insights into the effectiveness of the model and highlight areas for future improvement.

4.1 Experimentation

4.1.1 Experiment Setup

The proposed model was tested on a diverse set of coding scenarios extracted from our dataset. These scenarios were chosen to represent a wide range of programming constructs and complexities. The performance of the model was then compared against the simple rule-based system, our chosen baseline.

4.1.2 Evaluation Metrics

The system was evaluated based on the following metrics:

- Accuracy: The proportion of correct completions suggested by the system.
- Precision and Recall: Specifically important in cases where multiple correct completions are possible.
- Response Time: The time taken by the system to generate a completion suggestion.

4.2 Results

The analysis of the proposed method indicates that it did not perform as well as expected when compared to the simple rule-based system. In terms of accuracy and precision, the model was notably less effective, particularly in complex coding scenarios. This suggests that the system may have difficulties in accurately applying contextual constraints, leading to less relevant code completions. Furthermore, the recall of the model was also lower compared to the baseline, highlighting a potential issue in handling a diverse range of coding contexts and often missing correct completions.

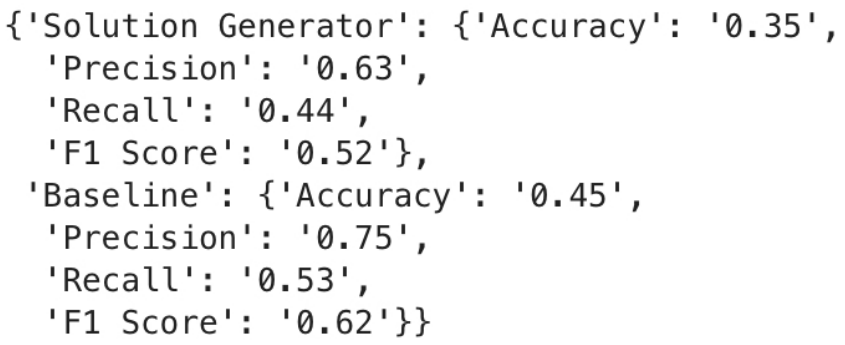

Figure 2: Evaluation results of the model compared with the baseline

In addition, the response time of the model was longer than that of the baseline. This longer response time is a point of concern, especially since it does not correspond with an improvement in completion suggestions. When benchmarked against oracle performance, the model fell significantly short of achieving human-like completions or approaching the theoretical maximum performance of an expressive classifier. This gap is larger than anticipated and underscores the need for substantial improvements in the CSP approach for code completion.

These results suggest that the current implementation of the CSP approach in the code-completion model is less effective than simpler, rule-based methods. The system's lack of accuracy, coupled with an increased response time, underscores critical areas that require substantial enhancement.

In conclusion, the performance of the model in its current state represents a drawback in code completion technology. The system struggles to compete with even the baseline models in terms of accuracy, recall, and efficiency. These findings highlight the necessity for significant improvements in understanding complex contexts and optimising response times. Future developments should aim to address these key issues, making the model a more viable and efficient tool in the realm of code completion.

5 Conclusion and Future Work

In concluding our work on the Constraint Satisfaction-Based Code Completion, it's crucial to acknowledge the outcomes and insights derived from this initiative. Tasked with the objective of applying constraint satisfaction principles to enhance code completion tools, the proposed method ultimately fell short in surpassing the efficacy of more basic rule-based systems. Although unexpected, this result has proven to be a significant opportunity for learning.

One of the primary challenges faced by the model was the complexity inherent in accurately modelling and solving constraint satisfaction problems within the dynamic and varied context of coding environments. The effectiveness of the system was also heavily dependent on the dataset used. Limitations in the dataset, particularly in terms of coverage and diversity, likely played a significant role in impacting the system's ability to learn and make accurate predictions. Furthermore, the model encountered substantial computational overheads due to its multi-layered approach, which included constraint ex-

traction, solution generation, and evaluation. These overheads affected both the response time and the practical usability of the system.

The project has highlighted several key lessons. It has underscored the gap that can exist between theoretical models and their practical implementations. It has also brought to light the fact that sometimes simpler models may be more effective, especially in scenarios where rapid response and computational efficiency are crucial. Additionally, the quality and comprehensiveness of the training dataset are paramount in data-driven approaches like the code-completion tasks.

Looking ahead, there are several avenues for improving the model. Simplifying and optimising the constraint models to better balance complexity and performance would be a vital first step. Expanding and diversifying the dataset to cover a broader range of coding scenarios and contexts could significantly improve the system's learning and adaptability. Considering hybrid approaches that integrate elements of rule-based systems with the model could capitalise on the strengths of both methodologies.

In conclusion, while the proposed method did not achieve its initial performance goals, it has highlighted the complexities involved in automating coding tasks and the delicate balance needed between theoretical models and practical applications.

References

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

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*, pages 444–463.

Li, J., Wang, Y., Lyu, M. R., and King, I. (2017). Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*.

Raychev, V., Bielik, P., and Vechev, M. (2016). Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

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*, pages 1433–1443.