Simple feedback system for programming assignment to improve idiomatic use of code

Master's Thesis submitted to the Constructor Institute
in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Software Engineering

presented by Flora Teklit

under the supervision of Prof. Manuel Oriol
co-supervised by Ilya Biametov

June 2023
I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Flora Teklit
Schaffhausen, 16 June 2023
Acknowledgements

I would like to express my sincere gratitude to the administration of Constructor Institute Schaffhausen, my thesis advisor Prof. Manuel Oriol, and co-advisor Ilya Baimetov. Their support, guidance, and expertise have been instrumental in the successful completion of this thesis.
Contents

List of Figures

Introduction

Literature Review

2.1 Literature review on programming Automatic feedback system for programming assignments

2.1.1 A detailed review of some papers

2.2 Literature review on a real-time suggestion for improving idiomatic use of a code

2.2.1 The Importance of Idiomatic Programming:

2.2.2 Real-Time Suggestion Generation:

2.2.3 Approaches of implementation

2.2.4 Evaluation and User Perception:

2.2.5 Conclusions

Approach

3.1 System overview

Implementation

4.1 Running the LLMs

4.1.1 OpenAI

4.1.2 Starcoder

4.2 Creating endpoint(API)

4.3 Script for web scraping

4.4 Implementation Challenges

Result and Analysis

5.1 Comprehensive understanding of idiomatic programming practices

5.2 Research and propose best possible approaches to design and implement a suggestion-generating system

5.3 Results achieved by each model
List of Figures

3.1 Proposed solution model .............................................. 14
3.2 Proposed solution model .............................................. 15
4.3 sample JS script to scrape webpages .............................. 19
4.4 sample js code pairs gained by web scraping ...................... 20
4.5 sample python code pairs gained by web scraping ............... 21
5.6 Non-idiomatic Python snippet 1 ................................... 24
5.7 Non-idiomatic Python snippet 2 ................................... 25
5.8 Non-idiomatic python snippet 3 ................................... 26
5.9 Non-idiomatic Javascript snippet 1 ................................. 27
5.10 Non-idiomatic Javascript snippet 2 ............................... 28
Abstract

The aim of this master’s thesis is to investigate the utilization of pre-trained large language models in the domain of programming education to enhance students’ programming skills, particularly focusing on the idiomatic use of code. We propose the development of an automated feedback generation system that harnesses the power of pre-trained language models, such as GPT-3 and starcoder, to provide personalized and actionable feedback to students on their code submissions. By analyzing the code syntactically, semantically, and through contextual understanding, the system will offer suggestions and recommendations to improve the idiomatic use of a code. To achieve this, the thesis explores and researches fitting models, the process of fine-tuning and adapting pre-trained language models to programming-specific contexts, Furthermore compares results achieved by the selected model and suggests future work to improve results. The findings of this research will contribute to the field of programming education by demonstrating the potential of pre-trained language models in providing tailored and constructive feedback on code submissions. This thesis will shed light on the benefits and limitations of leveraging these models, paving the way for future advancements in automated feedback systems and promoting the development of more idiomatic programming skills among students.
Introduction

In computer programming, idioms (best practices based on the language's syntax, features, and design patterns) are essential for writing code that adheres to language-specific conventions, best practices, and style guidelines. While the benefits of idiomatic programming in terms of code quality, readability, and maintainability and its impact on system efficiency are well-documented, there still requires further investigation to bring a way to help individuals to achieve this practice. Students often struggle to write code that follows best practices, adheres to language-specific conventions, and incorporates idiomatic patterns. This can lead to code that is difficult to read, understand, and maintain. Findings by Zejun Zhang et al. [11], which investigated a big dataset for Python, propose some actionable suggestions for using Python idioms. The existing methods of assessing programming assignments primarily focus on correctness, neglecting code quality, readability, and adherence to idiomatic practices. Thus, students develop poor coding habits or fail to appreciate the importance of writing code that aligns with industry standards and community norms. To address this challenge, this thesis researches possible and efficient approaches for developing an intelligent suggestion-generating system that analyzes students' code submissions and provides real-time feedback and suggestions for writing programming assignments in a more idiomatic manner. The proposed solution leverages pre-trained large language models (LLM) that can suggest a best practice in programming.

The key objectives of this thesis are as follows:

1. Comprehensive understanding of idiomatic programming practices: Conduct an extensive literature review and study industry practices to develop a comprehensive understanding of idiomatic programming practices in different programming languages, which includes language-specific conventions, design patterns, and coding style guidelines and their impact on code performance.

2. Research and propose the best possible approaches to design and implement a suggestion-generating system: investigate methods and customize them to a specific need Design and implement a system that can analyze students' code submissions and provide suggestions to improve the idiomatic use of a code snippet. The system mainly utilizes different pre-trained Large language models and compares the results.

3. Machine learning-based personalized suggestions: Explore and integrate machine learning techniques to enhance the suggestion-generating system's ability to provide personalized suggestions based on students' coding styles and experience levels. The system will learn from a large dataset of idiomatic code examples to provide contextually relevant and tailored suggestions for individual students.
4. Evaluation and assessment: evaluation of the proposed approaches by testing over a test set and comparing it with the expected results.

This research aims to contribute to the field of computer programming education by providing students with real-time, personalized feedback to enhance their understanding and application of idiomatic programming practices. By promoting code that is idiomatic, readable, and maintainable, the system will help bridge the gap between theoretical knowledge and practical coding skills, facilitating the development of proficient and industry-ready programmers.
Literature Review

Numerous tools and papers provide feedback for programming solutions submitted by students, each achieving different types of feedback based on different approaches. Below will be discussed and evaluated the related works of feedback generators for programming assignments based on:

- Type of feedback they provide,
- The method they used and level of instructors’ involvement
- Incremental effort required to add a new assignment
- How the code is analyzed

The literature review is presented below in two parts.

- A general research that presents state of the art in automatic feedback systems for programming to identify which area needs more investigation by evaluating the above 4 points for each paper.
- A specific research that presents state of the art in feedback systems to improve idiomatic use of a language.

2.1 Literature review on programming Automatic feedback system for programming assignments

General categorization of the papers reviewed based on the type of feedback they give includes, hints towards correct solutions and feedback on code refactoring, research results C. Douce, et al. Automatic test-based assessment of programming [6] and R. Romli, et al. Automatic programming assessment and test data generation [7] provide solutions for automatic assessment which reduces instructors’ involvement while grading by providing test cases, while the student only receives an instant response stating whether the code fails or passes the test. As an implementation approach, LLM and predefined model solutions with test cases are mostly used. Due to the impressive performance obtained using Large Pre-Trained Language Models (LLMs) on many code-related tasks, researchers have started to directly use LLMs. Chunqiu S. et al. Conversational Automated Program Repair [8] utilizes large language models to automatically generate patches for bugs.
2.1 Literature review on programming Automatic feedback system for programming assignments

2.1.1 A detailed review of some papers

1. The feedback services offered in Hieke Keuning, 2020 [1] are next-step hints, validation of steps, and showing complete solution paths. Rules and strategies have to be specified to provide these services.

   - Types of issues the feedback system covers, e.g. – code refactoring
   - How feedback is given – on demand with options to check the progress
   - How the code is analyzed – based on the IDEA framework (Interactive domain-specific exercise assistants)
   - Incremental effort required to add a new assignment – teacher provides annotated model solutions for each new question added

2. A tutor Hieke Keuning et al. 2014 [2] for imperative programming with incrementally constructing programs and Alex Gerdes et al., 2016 [9] derive feedback from an exercise description together with a set of model solutions. An instructor annotates the model solutions to adapt the feedback, such as providing specific feedback messages, allowing alternative statements, and enforcing the use of a specific language construct step-by-step incremental hints.

   - Types of issues the feedback system covers - hints towards the correct solution and alternative algorithms
   - How feedback is given – the detailed hint is given on demand, while hints for the next step are available in real-time
   - How the code is analyzed – it’s built on top of the IDEAS framework (Interactive domain-specific exercise assistants), which provides services to build exercise assistants to help students solve problems incrementally
   - Incremental effort required to add a new assignment – a teacher is required to provide an annotated solution model per each exercise
   - Not suitable for complex programs

3. The main functionality of the tutor Daniel L. Gauthier (2020) [3] is obtained by means of strategy-based model tracing with incremental development of different solutions in various forms to a programming exercise with automated feedback and teacher-specified programming exercises, solutions, and properties. The teacher provides annotated solution models and test cases.

   - Types of issues the feedback system covers – hints toward correctness
   - How feedback is given – the hint is given on demand
   - How the code is analyzed – it integrates different tools, it uses error detection and message display of helium compiler, annotated feedback from solution models provided by a teacher, test cases that verify predefined property provided by the teacher
   - Incremental effort required to add a new assignment – a teacher is required to provide an annotated solution model per each exercise
4. The work on Georgiana Haldeman 2021[4] explores techniques for extending an auto grader to provide corrective and formative feedback on programming assignment submissions using a mixed approach. CSF2’s methodology focuses on linking errors in student programs to concepts and skills required for solving a programming assignment. It is data-driven and it relies on collecting and analyzing assignment submissions to generate hints that can be used during future semesters.

- Types of issues the feedback system covers, e.g. – correctness
- How feedback is given – e.g. after submission
- How the code is analyzed – it uses a data-driven method, where a large set of submission data is analyzed
- Incremental effort required to add a new assignment – a teacher is required to provide a test case and solution references with error feedback that are considered to happen most frequently.

5. Jialu Zhang et al. 2022[27] use Large Language Models to Repairing Bugs in Python Assignments

- Types of issues the feedback system covers - provides syntactic and semantic feedback
- How feedback is given – on demand
- How the code is analyzed – uses a Large language model trained for coding
- Incremental effort required to add a new assignment – a teacher is supposed to give only the question.

The state-of-the-art research shows that real-time suggestions for the correctness of a code and bug fixes are widely researched. But, as for readability, the best practice of programming and conventions still requires further investigation.

2.2 Literature review on a real-time suggestion for improving idiomatic use of a code

Idiomatic programming assignments require students to write code that adheres to established best practices, conventions, and idiomatic patterns of the programming language. Real-time suggestion generation systems have emerged as a promising approach to provide students with immediate feedback and guidance for writing code in a more idiomatic manner. This part of the literature review aims to explore the existing research on enhancing idiomatic programming assignments through real-time suggestion generation and the impact of idiomatic code on the efficiency of a system.

2.2.1 The Importance of Idiomatic Programming:

Numerous studies highlight the significance of idiomatic programming in code quality, readability, maintainability, and collaboration among developers. P. Leelaprute et al. 2022[17] emphasize that idiomatic code improves software quality, facilitates code reuse, and enhances collaboration. Zhang et al. (2018)[18] discuss the benefits of idiomatic programming in code
2.2 Literature review on a real-time suggestion for improving idiomatic use of a code comprehension, reducing the learning curve, and increasing productivity. Ahmad Tahmid 2020 \cite{19} emphasizes the impact of idiomatic programming on software maintenance, debugging, and code refactoring.

2.2.2 Real-Time Suggestion Generation:
Real-time suggestion generation systems aim to provide students with immediate feedback and suggestions for improving the idiomatic use of code. These systems leverage techniques from natural language processing, programming language analysis, and machine learning to identify and suggest improvements based on the code’s syntax, style, and idiomatic patterns. They often analyze code in real-time, highlighting areas where the code can be enhanced to align with idiomatic practices.

2.2.3 Approaches of implementation
Various approaches have been explored in real-time suggestion generation systems for idiomatic programming. Most of the approaches are language specific largely on working on Python. Et al. Zejun Zhang 2022 define the syntactic patterns for detecting nonidiomatic code for each Pythonic idiom and devise atomic AST-rewriting operations and refactoring steps to refactor non-idiomatic code into idiomatic code. This work tested over 4,115 refactorings applied to 1,065 Python projects from GitHub which confirmed the high accuracy, practicality, and usefulness of the tool on real-world Python code. The paper by Veselin Raychev et al. \cite{26} proposed an approach based on program synthesis and transformation rules to automatically refactor non-idiomatic code snippets.

2.2.4 Evaluation and User Perception:
The evaluation of real-time suggestion generation systems for idiomatic programming involves assessing their effectiveness, usefulness, and impact on students’ learning outcomes. Studies often employ quantitative measures, such as code quality metrics and completion time, as well as qualitative methods, such as user surveys and interviews, to gather feedback from students and instructors. Evaluation results generally demonstrate positive user perception and improved idiomatic use of code with the use of real-time suggestion generation systems.

Real-time suggestion generation systems hold significant promise for enhancing idiomatic programming assignments. The existing literature highlights the importance of idiomatic programming, the benefits of real-time suggestion generation, and the challenges associated with developing such systems.

2.2.5 Conclusions
Real-time suggestion generation systems hold significant promise for enhancing idiomatic programming assignments. The existing literature highlights the importance of idiomatic programming, the benefits of real-time suggestion generation, and the challenges associated with developing such systems.
Approach


3.1 System overview

The main idea in this approach is to leverage pre-trained Large language models that are specific for programming. Some of the models require further fine-tuning on a dataset that is specifically designed on generating idiomatic versions of a code snippet. Having this as the main function of this thesis, an API is also available to make this function easily available to use.
Pretrained Language Models (PLMs) work by leveraging large-scale datasets of code to learn the statistical patterns and structures present in programming languages. These models are pre-trained on a diverse range of code repositories, which allows them to capture the syntax, semantics, and idiomatic constructs of various programming languages.

The training process involves predicting the next token in a sequence of code given the preceding tokens. By learning from vast amounts of code data, PLMs develop an understanding of the relationships between different tokens, the context in which they appear, and the overall structure of programs. During the training phase, PLMs utilize a transformer architecture, which consists of multiple layers of self-attention and feed-forward neural networks. Self-attention mechanisms help the model capture dependencies between different parts of the code, allowing it to effectively process and understand long-range dependencies. The model's parameters are updated through a process called back propagation, where the model's predicted outputs are compared to the ground truth code, and the gradients are used to adjust the parameters.

Once pre-trained, these PLMs can be fine-tuned on specific code generation tasks or downstream applications. Fine-tuning involves training the model on a smaller, task-specific dataset to adapt it to a particular task.

The process of further fine-tuning pre-trained models requires a large and well-analyzed dataset to produce the required outputs. Finding and restructuring such a dataset was a bottleneck in this work.

The steps of fine-tuning a pre-trained LLM model

1. Fine-tuning for idiomatic code generation: After the initial pretraining, the LM is fine-tuned on a task-specific dataset that includes pairs of non-idiomatic code and their corresponding idiomatic versions. The fine-tuning process encourages the model to learn to generate code snippets that are more idiomatic based on the provided examples.
2. Encoding input code: To generate an idiomatic version of a given code input, the input code is encoded as a sequence of tokens that the LM can process. This encoding typically involves tokenizing the code into smaller units such as functions, variables, and language keywords.

3. Generation process: The LM takes the encoded input code as a prompt and generates the next token or sequence of tokens to complete the code. The generation process involves sampling from the predicted probability distribution of tokens, taking into account the context of the code input and the learned patterns from the pre-trained and fine-tuned models.

4. Post-processing and validation: The generated code snippets are typically post-processed and validated to ensure syntactic correctness and adherence to idiomatic practices. This step involves reviewing the generated code, performing code analysis, and making any necessary adjustments or modifications to improve the idiomatic use of a code and address potential errors or inefficiencies.

Using the pre-trained large language models as default without fine-tuning also follows the same steps except it does not include further training on a dataset.

Figure 3.2. Proposed solution model
Implementation

The codebase is Python and javascript. The implementation consists of three parts.

4.1 Running the LLMs

The first step in this section was to choose a fitting model. After careful trial and investigation of many LLMs for coding, two fitting models are selected to proceed with the work. The models are openAI and starcoder. This work engineered multiple prompts for both LLMs to evaluate which generated better explanations. This prompt engineering process involved trial and error to determine which prompt phrases and level of detail resulted in the best responses from the LLM. Although codeBERT (a bimodal pre-trained model for a programming language (PL) and natural language (NL)) was a potential pre-trained model, it needs to be further fine-tuned on a specific dataset to generate a better idiomatic way of writing a code snippet. These pre-trained LLMs being the backbone of this project, Hugging Face (a large open-source community that builds tools to enable users to build, train, and deploy machine learning models based on open-source code and technologies) is the main source of the models for this work. The code base for running these models is mainly python.

4.1.1 OpenAI

Different versions of openAI are used.

- Text-davinci-002,
- Code-davinci-edit-001: a GPT-3.5 models (can understand and generate natural language or code) that is optimized for code-completion tasks.

4.1.2 Starcoder

A State-of-the-Art open source LLM for Code, Code LLMs trained on permissively licensed data from GitHub, including from 80+ programming languages, Git commits, GitHub issues, and Jupyter notebooks. It trained a 15B parameter model for 1 trillion tokens and fine-tuned from the StarCoderBase model for 35B Python tokens.

StarChat: a specialized version of StarCoder that has been fine-tuned on the Dolly and OpenAssistant datasets. It is a 16-billion parameter model that was pre-trained on one trillion tokens sourced from 80+ programming languages, GitHub issues, Git commits, and Jupyter notebooks. This model requires the GPU to run it locally. So an inference endpoint provided by hugging face is used.
4.2 Creating endpoint (API)

The end point is created with Flask in Python. It takes a code snippet as a param and returns
the better idiomatic way of writing that code snippet.

4.3 Script for web scraping

Javascript is used to write scripts for web scraping that have idiomatic coding examples. The
dataset for validation is collected by web page scraping that has idiomatic coding practices. The
script for web scraping is written in javascript. The main source is [16].
import jsdom from "jsdom";
import fetch from "node-fetch";
const url = "https://programming-idioms.org/cheatsheet/Java";
import fs from 'fs';
import { client } from "@radio/client";

async function getContent() {
    const response = await fetch(url);
    if (response.ok) {
        let content = await response.text();
        const parser = new jsdom.JSDOM(content);
        const table = parser.window.document.querySelector('table');
        const tBody = table.tBodies;
        const lines = tBody.item(0).querySelectorAll('.cheatsheet-line');

        let QandA = {};
        let index = 0;
        let all_codes=[]
        lines.forEach((line) =>{
            let question = line.querySelector('.idiom-title a').textContent.trim();
            /\* add question */
            QandA[index] = { index, question, codes: []}

            let codes = line.querySelectorAll('code');
            codes.forEach((code)=>{
            QandA[index].codes.push(code.textContent);
            if(index == 1){
                // console.log("idiom", code.textContent);
            }
        });
        all_codes.push( QandA[index].codes)
        index++;
    });
    exportArrayToCSV(all_codes,'export_java.csv');
}

Figure 4.3. sample JS script to scrape webpages
Non Idiomatic code

```javascript
for (let i = 0; i < 10; i++) {
    console.log(Hello*);
}

var bli = function() {
    console.log(Hello World!!!);
}

function square(x) {
    return x * x;
}

var p = { x: 1.122, y: 7.45 };

items.forEach(x => {
    doSomething(x);
});

items.forEach((val, idx) => {
    console.log(index="" + idx + "

});

const x = {one: 1, two:2}

x.set("two*

class Node {
    constructor (data) {
        this.data = data
        this.left = null
        this.right = null
    }
}

for (var i = 0; i < 10; i++) {
```

Figure 4.4. sample js code pairs gained by web scraping
4.4 Implementation Challenges

There are a set of challenges faced while Fine-tuning the pre-trained models.

1. Dataset Size and Quality: Fine-tuning a pre-trained model requires a task-specific dataset. Obtaining a sufficiently large and high-quality dataset that is specific to our needs, a dataset of idiomatic and non-idiomatic code pairs is challenging. Limited data leads to overfitting, where the model fails to generalize well to unseen examples.

2. Computational Resources: Fine-tuning the large-scale models, which are transformer-
based architectures, is computationally demanding and requires substantial computational resources, including powerful GPUs or specialized hardware. Limited access to such resources hinders the fine-tuning process.

3. Getting the perfect fit model: The open source models for programming (codeBERT, codegen, codeParrot) are mostly designed for generating code based on an instruction not to provide feedback for existing code.
Result and Analysis

This result and analysis part discusses the key objectives of this thesis as mentioned in the introduction section.

5.1 Comprehensive understanding of idiomatic programming practices

Idiomatic programming practices refer to the use of programming language features, patterns, and conventions that are considered standard and recommended by the language’s community. These practices aim to enhance code readability, maintainability, and efficiency. The benefits of idiomatic programming practices, including improved readability, maintainability, and performance, make it worthwhile to invest time and effort in learning and adopting these practices. By adhering to idiomatic practices, developers contribute to a more consistent and efficient codebase and enhance their skills as a developer. Continuously improving one’s understanding and implementation of idiomatic practices will have a positive impact on your code quality and collaboration with other developers.

5.2 Research and propose best possible approaches to design and implement a suggestion-generating system

Considering the effort it takes to implement, accuracy, and inclusiveness of more than one programming language, pre-trained LLMs for programming are selected as an approach. Using the pre-trained models reduces the effort to train a new model. The target programming languages are python and javascript. Star coder and openAI are the best fitting models for this work (to provide a better idiomatic way of writing a code snippet). CodeBERT is also well leveraged for other programming tasks by many researchers, but coming to our specific need it needs a lot of customization. An API to access this work is also provided.

5.3 Results achieved by each model

To validate how accurate the results achieved are, it is tested on idiomatic and non-idiomatic code snippet pairs. These code snippet pairs are taken from [16] (which provides idiomatic and non-idiomatic versions of a code in different programming languages). Below will be presented the 2 Python code snippets and 2 javascript code snippets inputs and the corresponding results.
Non-idiomatic code

cities = ['Nairobi', 'Kampala', 'Lagos']
index = 0
while index < len(cities):
  print(cities[index])
  index += 1

Text-davinci-002

Idiomatic Code: - for city in cities: print(city)

Suggestions: - 1. Use a for loop instead of a while loop: for city in cities: print(city) 2. Use the enumerate function to keep track of the index: for index, city in enumerate(cities): print(index, city)

Code-edit-davinci

code:
cities = ['Nairobi', 'Kampala', 'Lagos'] for city in cities: print(city)

Startcoder

cities = ["Nairobi", "Kampala", "Lagos"]
for idx, city in enumerate(cities):
  print("{}: {}")

This will loop through each item in cities and print out its index along with the value. The {0} is replaced by the current index while {1} gets replaced by the corresponding element from the list. So for example if we have three elements ["a", "b", "c"] then this would output
<table>
<thead>
<tr>
<th><strong>Non-idimatic code</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>file = open('data.txt', 'r')</code></td>
</tr>
<tr>
<td><code>content = file.read()</code></td>
</tr>
<tr>
<td><code>print(content)</code></td>
</tr>
<tr>
<td><code>file.close()</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Text-davinci-002</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Idiomatic Code: -</td>
</tr>
<tr>
<td><code>with open('data.txt', 'r') as file:</code></td>
</tr>
<tr>
<td><code>content = file.read()</code></td>
</tr>
<tr>
<td><code>print(content)</code></td>
</tr>
</tbody>
</table>

Suggestions: -
- Replace `file.read()` with `file.readlines()` for improved readability
- Use the with statement to automatically close the file when finished

<table>
<thead>
<tr>
<th><strong>Code-davinci-edit-001</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Code:</td>
</tr>
<tr>
<td><code>with open('data.txt') as file:</code></td>
</tr>
<tr>
<td><code>content = file.read()</code></td>
</tr>
<tr>
<td><code>print(content)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Starcoder</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>with open(&quot;data.txt&quot;, &quot;r&quot;) as f:</code></td>
</tr>
<tr>
<td><code>content = f.read()</code></td>
</tr>
<tr>
<td><code>try:</code></td>
</tr>
<tr>
<td><code>print(content)</code></td>
</tr>
<tr>
<td><code>except Exception as e:</code></td>
</tr>
<tr>
<td><code>+ print(f&quot;Error reading file: {e}&quot;)</code></td>
</tr>
<tr>
<td><code>finally:</code></td>
</tr>
<tr>
<td><code># Make sure to close the file after finishing</code></td>
</tr>
<tr>
<td><code>f.close</code></td>
</tr>
</tbody>
</table>

This uses the built-in `open()` function to open the file in read mode (using the "r" argument). Then, it uses a context manager (with) to automatically close the file when the block of code finishes executing. Inside the block, the contents of the file are stored in the variable `content`.  

Figure 5.7. Non-idimatic Python snippet 2
## Non-idiomatic code

```python
def check_even(numbers):
    has_even_number = False
    for num in numbers:
        if num % 2 == 0:
            has_even_number = True
            break
    print(has_even_number)
```

## Idiomatic Code: Text-davinci-002

```python
numbers = [1, 2, 3, 4, 5]
has_even_number = any(num % 2 == 0 for num in numbers)
print(has_even_number)
```

Suggestions:
1. Use a boolean flag to track if an even number has been found, instead of using a break statement.
2. Iterate over the numbers in reverse order.

## Code-davinci-edit-001

```python
numbers = [1, 2, 3, 4, 5]
has_even_number = False
for num in numbers:
    if num % 2 == 0:
        has_even_number = True
        break
print(has_even_number)
```

This is Python idiomatic way of writing the same logic above. The `any()` function is used to check if any number in the list is even.

## Starcoder

One possible improvement could be to use the `all()` function instead of writing your own conditional statement inside the loop. It works like this:

```python
numbers = [1, 2, 3, 4, 5]
if all([num % 2 == 0 for num in numbers]):
    print("No even numbers found!")
else:
    print("At least one even number was found.")
```

In this case, `all()` takes a list or tuple of expressions and returns true only if all those expressions evaluate to true. Since we're checking whether every number in the list is odd, we simply need to negate the condition (i.e., make it return false if at least one number is even).

---

**Figure 5.8. Non-idiomatic python snippet 3**
Non-idiomatic code

```javascript
function add(a, b) {
    return a + b;
}
console.log(add(5, 3));
```

Text-davinci-002

Ilidiomatic Code:
```javascript
const add = (a, b) => a + b;
console.log(add(5, 3));
```

Suggestions: 
- Declare the function with the keyword "function" followed by a unique function name. 
- Add parameters inside the parentheses of the function declaration, separating multiple parameters with commas. 
- Use the keyword "return" followed by the value you want to return. 
- End your function declaration with a semicolon.

Code-davinci-edit-001

```javascript
function add(a, b) {
    return a + b;
}
console.log(add(5, 3));
function addN (n) {
    return function (a) { return a + n; };
} 
var three = addN(3)
console.log(addN(5)(10))
```

Starcoder

```javascript
const add = (a,b) => a+b; // arrow functions can be used to shorten your function definitions and remove unnecessary braces {}
console.log(add(5,3)) // Output: 8
```

This is just one possible approach for writing that same function. You may find other ways to do it depending on personal preference or coding style preferences.

Figure 5.9. Non-idiomatic Javascript snippet 1
The results of the experiment show that pre-trained large language models can be efficiently
5.3 Results achieved by each model

used to generate feedback on how to improve the idiomatic use of a code.

- **Text-davinci-002**, gives accurate results and not only provides a more idiomatic version of the code but also is able to give a suggestion on how to improve the code, which would be handier to improve students’ skill.

- **Code-davinci-edit-001**, gives an accurate result too. It's able to generate an idiomatic version of a code snippet. But it is not as good as the Text-davinci-002 model in providing suggestions on how to improve the code.

- **Starcoder** also achieves the required result. The suggestions generated are more on the concept of what to be improved and in some cases it provides code snippet examples that are not in the input.
5.3 Results achieved by each model
Conclusion and Future Work

6.1 Limitation and Future work

Despite the achievements and contributions of this thesis, there are several limitations that should be acknowledged. These limitations present opportunities for future research and enhancements to the automated feedback generation system.

1. Lack of successful fine-tuning: The current implementation of the system does not include the fine-tuning of pre-trained language models specifically for programming education. Even tho fine-tuning codeBERT was tried, it didn't give the required result as it needs big and quality data. Fine-tuning models on domain-specific datasets could potentially improve the system's ability to provide more accurate and contextually aware feedback. Future work should explore fine-tuning methodologies tailored to programming contexts, taking into account idiomatic code practices, common programming patterns, and specific programming languages.

2. Limited scope of analysis: The system's analysis is currently focused on the syntactic and semantic aspects of code. While it provides feedback on idiomatic usage, it may not capture other important factors such as performance optimization, security considerations, or design patterns. Future research could expand the system's capabilities to encompass a broader range of code quality aspects, providing comprehensive feedback to students for further improvement.

3. Dataset limitations: The quality and diversity of the dataset used for training and evaluation can impact the system's performance. The availability of larger and more diverse datasets of idiomatic and non-idiomatic code examples would enhance the system's ability to generalize and provide accurate feedback on various programming styles and practices.

4. Contextual understanding challenges: pre-trained language models may face challenges in fully understanding the context-specific requirements of different programming tasks. Future work could explore techniques to incorporate task-specific context information into the feedback generation process, allowing the system to provide more tailored and context-aware suggestions to students.

5. Evaluation across different programming languages: The current evaluation of the system has focused on a specific programming language or a limited set of languages. Extending the evaluation to encompass a broader range of programming languages would provide insights into the system's effectiveness and applicability across diverse programming paradigms.
6. Integration with existing educational platforms: Integrating the automated feedback generation system with popular programming education platforms and tools would facilitate its adoption by educators and students. Future work should explore the development of APIs or plugins that seamlessly integrate the system’s functionality into existing programming learning environments.

7. Ethical considerations: As with any automated system, it is crucial to address potential biases, fairness, and ethical concerns. Future research should investigate methods to ensure that the automated feedback system is unbiased, fair, and does not reinforce any undesirable programming practices or biases.

In conclusion, while this thesis has demonstrated the potential of leveraging pre-trained language models for automated feedback generation on idiomatic code usage, there are several avenues for future work and improvements. By addressing the limitations and pursuing these future directions, researchers can further enhance the system’s capabilities, broaden its applicability, and contribute to the advancement of programming education by empowering students to write more idiomatic and high-quality code.

6.2 Conclusions

This thesis has explored the potential of leveraging pre-trained large language models to generate feedback for students, with a specific focus on improving their skills in writing idiomatic code. Through the development of an automated feedback generation system, we have demonstrated how these models can provide valuable insights and suggestions for enhancing idiomatic way of writing a code.

Throughout the research process, we have utilized pre-trained large language models to analyze code and offer feedback on areas where students can improve their code’s idiomatic usage. The system has showcased the capability of these models to understand the syntactic and semantic aspects of code, allowing for personalized feedback tailored to individual students’ needs.

The evaluation of the automated feedback generation system has indicated promising results. While the current implementation focused on utilizing pre-trained models without further fine-tuning, it is important to acknowledge that the performance of the system can be enhanced through fine-tuning techniques. Additionally, as pre-trained language models continue to advance and new models emerge, further investigations can explore the use of these models to improve automated feedback generation. Newer models may exhibit improved code understanding capabilities and allow for more nuanced feedback on specific programming concepts, idioms, and patterns.

Overall, this thesis contributes to the field of programming education by showcasing the benefits of leveraging pre-trained large language models to facilitate the learning process and encourage the development of idiomatic programming skills. The automated feedback generation system holds promise for educators and students, offering a scalable and efficient approach to enhance programming education and promote best practices in code development.

As technology continues to evolve, it is essential to adapt and refine automated feedback systems to keep pace with the changing needs of programming education. By harnessing the power of pre-trained language models and continuously improving their fine-tuning methodologies, we can create more sophisticated and context-aware systems that provide students with
targeted feedback, foster their growth as programmers, and equip them with the skills needed to excel in the dynamic world of software development.
## Abbreviation

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLM</td>
<td>large language model</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>IDEA</td>
<td>Interactive domain-specific exercise assistants</td>
</tr>
<tr>
<td>PLMs</td>
<td>Pretrained Language Models</td>
</tr>
<tr>
<td>PL</td>
<td>programming language</td>
</tr>
<tr>
<td>NL</td>
<td>natural language</td>
</tr>
</tbody>
</table>
Bibliography


```python
file = open('data.txt', 'r')
content = file.read()
print(content)
file.close()
```

```python
fruits = ['apple', 'banana', 'orange']
index = 0
for fruit in fruits:
    print(f'Index: {index}, Fruit: {fruit}')
    index += 1
```

```python
fruits = ['apple', 'banana', 'orange']
for index, fruit in enumerate(fruits):
    print(f'Index: {index}, Fruit: {fruit}')
```

```python
# Using a traditional for loop to create a new dictionary
keys = ['a', 'b', 'c']
values = [1, 2, 3]
ew_dict = {}
for i in range(len(keys)):
    new_dict[keys[i]] = values[i]
print(new_dict)
```

```python
keys = ['a', 'b', 'c']
values = [1, 2, 3]
new_dict = {key: value for key, value in zip(keys, values)}
print(new_dict)
```

```python
numbers = [1, 2, 3, 4, 5]
has_even_number = False
for num in numbers:
    if num % 2 == 0:
        has_even_number = True
print(has_even_number)
```

```python
numbers = [1, 2, 3, 4, 5]
has_even_number = any(num % 2 == 0 for num in numbers)
print(has_even_number)
```
| break                              | const numbers = [1, 2, 3, 4, 5];
| print(has_even_number)            | const clonedNumbers = [...numbers];
|                                   | console.log(clonedNumbers);
| var numbers = [1, 2, 3, 4, 5];     | console.log(clonedNumbers); |
| var clonedNumbers = [];           | const clonedNumbers = [...numbers];
| for (var i = 0; i < numbers.length; i++) { | console.log(clonedNumbers); |
|     clonedNumbers.push(numbers[i]); | const clonedNumbers = [...numbers];
| }                                 | console.log(clonedNumbers);
| console.log(clonedNumbers);       | const clonedNumbers = [...numbers];
|                                   | console.log(clonedNumbers);
| var person = { name: 'John', age: 30 }; | const person = { name: 'John', age: 30 }; | console.log(clonedNumbers); |
| var name = person.name;           | const { name, age } = person;                                |
| var age = person.age;             | console.log('Name: ' + name + ', Age: ' + age); |
| console.log('Name: ' + name + ', Age: ' + age); | const person = { name: 'John', age: 30 }; | console.log(clonedNumbers); |
|                                   | const { name, age } = person;                                |
|                                   | console.log('Name: ' + name + ', Age: ' + age); |
| var numbers = [1, 2, 3, 4, 5];     | const numbers = [1, 2, 3, 4, 5]; const doubledNumbers = numbers.map(num => num * 2);
| var doubledNumbers = [];          | console.log(doubledNumbers.join(','));                       |
| for (var i = 0; i < numbers.length; i++) { | console.log(doubledNumbers.join(',')); |
|     doubledNumbers.push(numbers[i] * 2); | console.log(doubledNumbers.join(',')); |
| }                                 | console.log(doubledNumbers.join(','));                       |
| console.log(doubledNumbers.join(',')); | console.log(doubledNumbers.join(',')); |
| console.log(doubledNumbers.join(',')); | console.log(doubledNumbers.join(',')); |
| console.log(doubledNumbers.join(',')); | console.log(doubledNumbers.join(',')); |
String name = "John"; String greeting = "Hello, " + name + "!";
System.out.println(greeting);

String name = "John";
StringBuilder greeting = new
StringBuilder();
greeting.append("Hello, ");
greeting.append(name).append("!");
System.out.println(greeting.toString());

int[] numbers = {1, 2, 3, 4, 5};
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}

int[] numbers = {1, 2, 3, 4, 5};
for (int number : numbers) {
    System.out.println(number);
}

FileInputStream fis = null;
try {
    fis = new FileInputStream("data.txt");
    // Code to read and process the file
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (fis != null) {
        try {
            fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

try (FileInputStream fis = new FileInputStream("data.txt")) {
    // Code to read and process the file
} catch (IOException e) {
    e.printStackTrace();
}