Verification and Validation Report: Software Engineering

Team 2, SyntaxSentinals
Mohammad Mohsin Khan
Lucas Chen
Dennis Fong
Julian Cecchini
Luigi Quattrociocchi

March 10, 2025

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Symbols, Abbreviations and Acronyms

symbol	description
T	Test

[symbols, abbreviations or acronyms – you can reference the SRS tables if needed —SS]

Contents

List of Tables

List of Figures

This document ...

3 Functional Requirements Evaluation

4 Nonfunctional Requirements Evaluation

4.1 Usability

4.2 Performance

4.3 etc.

5 Comparison to Existing Implementation

5.1 Overview of Existing Solutions

5.1.1 MOSS (Measure of Software Similarity)

MOSS is currently the standard tool for detecting code plagiarism in academic settings. Developed at Stanford University, it works by:

- Tokenizing code into a sequence of symbols
- Using document fingerprinting to detect similar code segments
- Generating a similarity score based on matching sequences
- Providing a web interface to view overlapping code segments

5.1.2 Other Notable Solutions

- **JPlag**: Uses a token-based approach similar to MOSS, but with different tokenization strategies
- **PLAGGIE**: Java-specific plagiarism detection tool used in academic environments
- **CodeMatch**: Commercial solution that uses both tokenization and metric-based methods

5.2 Our Implementation vs. Existing Solutions

5.2.1 Current Approach

Our implementation leverages CodeBERT, a pre-trained model for programming language understanding, to perform code pair comparison. Key aspects include:

- Utilization of transformer-based neural networks to understand code semantics
- Direct comparison of code pairs to determine similarity
- A modern, user-friendly interface for result visualization and analysis

5.2.2 Functional Comparison

Feature	**MOSS**	**Our CodeBERT Implementation**
Detection Method	Token-based fingerprinting	Neural representation and semantic understanding
Language Support	Multiple languages	Multiple languages (supported by CodeBERT)
Semantic Understanding	Limited (syntax-focused)	Enhanced (captures semantic meaning)
Resistance to Obfuscation	Low-moderate	Potentially higher
Speed	Fast for large submissions	Currently slower for large-scale comparisons
Visualization	Basic web interface	Modern, interactive UI

Table 1: Comparison between MOSS and our CodeBERT implementation

5.3 Advantages of Our Approach

1. **Semantic Understanding**: Unlike MOSS's purely syntactic approach, CodeBERT can potentially understand the meaning behind code, making it more difficult to fool with non-functional additions.

2. **Context Awareness**: Our implementation considers the context in which code appears, potentially reducing false positives from common solutions to standard problems.

3. **Modern Interface**: Our user interface provides a more intuitive experience for instructors reviewing potential plagiarism cases.

5.4 Current Limitations and Trade-offs

1. **Computational Requirements**: Neural models like CodeBERT require more computational resources than traditional approaches like MOSS.

2. **Development Maturity**: MOSS has been refined over decades, while our solution is still in the early stages of development.

3. **Validation**: Our approach needs more extensive testing across different programming languages and assignments to prove its effectiveness.

4. **Explainability**: Neural network decisions can be less transparent than token-matching approaches, making it potentially harder to explain why certain code pairs were flagged.

5.5 Design Decision Justification

1. **Choice of CodeBERT**: We selected CodeBERT over other models because it was specifically pre-trained on code from multiple programming languages, making it well-suited for understanding code semantics across different languages used in academic settings.

2. **Focus on Pair Comparison**: While MOSS compares each submission against all others, our current approach focuses on direct pair comparison, allowing for more detailed analysis of potentially plagiarized code.

3. **UI Priority**: We invested in a high-quality UI early in development to facilitate easier testing and validation of our detection results.

5.6 Future Improvements

1. **Hybrid Approach**: Combining neural understanding with traditional token-based methods to leverage strengths of both approaches.
2. **Performance Optimization**: Improving the computational efficiency to handle large class submissions more effectively.
3. **Tunable Sensitivity**: Implementing adjustable thresholds for different assignment types and programming languages.

6 Unit Testing

7 Changes Due to Testing

[This section should highlight how feedback from the users and from the supervisor (when one exists) shaped the final product. In particular the feedback from the Rev 0 demo to the supervisor (or to potential users) should be highlighted. —SS]

8 Automated Testing

9 Trace to Requirements

10 Trace to Modules

11 Code Coverage Metrics

References

Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Reflection.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

2. What pain points did you experience during this deliverable, and how did you resolve them?

3. Which parts of this document stemmed from speaking to your client(s) or a proxy (e.g. your peers)? Which ones were not, and why?

4. In what ways was the Verification and Validation (VnV) Plan different from the activities that were actually conducted for VnV? If there were differences, what changes required the modification in the plan? Why did these changes occur? Would you be able to anticipate these changes in future projects? If there weren't any differences, how was your team able to clearly predict a feasible amount of effort and the right tasks needed to build the evidence that demonstrates the required quality? (It is expected that most teams will have had to deviate from their original VnV Plan.)