Final Report

Name	Claire (Yumeng) Luo	Sang Jun Chun
UNI	yl4655	sc4658
Team Name	Imposters2	

1. Synopsis

Overview

Code similarity detection is used in plagiarism check, defect detection, etc. In this project, we are interested in utilizing code similarity detection in a particular use case: Detecting plagiarism in school projects.

We used 3 types of similarity detection tools: source code similarity detection tools, binary code similarity detection tools and text diff tools to detect plagiarism in individual coding assignments for COMS4156 and evaluate the performances of these tools.

Novelty

Code similarity detection is useful for plagiarism check, defect detection, and more. But, there are multiple approaches to detecting code similarity, and it is unclear which type of tools would perform most optimally for a given task. In this project, we want to examine the effectiveness of using code similarity detection tools particularly for detecting plagiarism in school programming projects.

Readers concerned with programming plagiarism in general, and wishing to manually incorporate a code similarity tool to examine different works, will find our project particularly interesting, as it will provide empirical data for each tool's performance, and learn specific tools best fit for the task.

Value to our user community

While this project specifically focuses on plagiarism check for modest-sized academic work, it can be applied to plagiarism detection in other domains, such as in business (e.g. online coding interview), as well. Furthermore, it would also be applicable for researchers hoping to develop or improve code similarity detection tools for academic plagiarism. In that sense, while this project will target academic instructors, who are the most likely to use code similarity detection tools for checking plagiarism, as the core audience, the project should provide meaningful values to researchers and non-academic audiences as well.

The audience will obtain from this project the empirical evaluation of different types of code similarity detection tools: source code similarity detection, binary code similarity detection, and text difference detection. The paper will provide by how much a particular tool outperforms another tool, and prescribe the most effective tool for detecting plagiarism in academic code, which the audience will find useful.

2. Research Questions

Explain in detail your **research questions** and their answers. These should include evaluating your work in comparison to other work with **specific examples**, **methodology**, **metrics**, **results and findings**. (Part 2 is the meat of your final report and should be several pages.)

To examine the effectiveness of the similarity detection tools, we propose 3 research questions focusing on different aspects of this problem. The questions are namely:

1. **RQ1:** How effectively can code similarity tools detect **different degrees of code plagiarism**? (i.e., are the tools susceptible to producing false negative results?)
2. **RQ2:** How do different types of code similarity tools react to **original works** addressing the same problem? (i.e., are the tools susceptible to producing false positive results?)
3. **RQ3:** Of the approaches examined, **which is particularly effective** for detecting academic plagiarism in code? (i.e, the overall best performance)

To answer the above questions, we conducted experiments over 4 different similarity detection tools using 37 real world coding assignments and 5 artificially plagiarized assignments. The next section explains in detail the methodology used.

Methodology

Dataset

We chose the individual java coding assignments for COMS 4156 as our data. This assignment builds the backend of an online tic-tac-toe game, and averages about 500-600 lines long across 5 different files.

We collected 37 meaningful submissions from the fork list and shuffled and masked the student names into ids. A copy of such 37 submissions can be found in our git repository (found here).

Due to no guarantee of finding plagiarization among the 37 samples, we created 5 additional samples which are plagiarizations of each other by different types of modifications.(found here)

Additionally, the 2 binary similarity detection tools require a training phase. Since we only have access to 37 data, further splitting such a dataset is impractical. Therefore, for these 2 tools, we used the training set provided by the author of the tools to train and used our gathered dataset for testing. (training set found here and here)

To perform pairwise comparison, we generated 37*36 /2 = 666 pairs of distinct code. For each distinct pair of samples, we manually generate a binary label using “0” for “non plagiarized pair” and “1” for “plagiarized pair”.

The standard used in determining the plagiarization state is “if two code pairs can be converted into each other with simple variable/function refactor, structure change then they are a plagiarized pair”.

Each function in each code has been first summarized into key features as listed in the second tab in this document. Pairs with more than 1 similar function are determined as a true pair.

Metric

We evaluated the 666 pairs of codes over 4 different similarity detection tools, namely: DeepSim, Gemini, Plaggie, SIM. These tools require different forms of input therefore some preprocessing is needed. See Appendix 7.0 for the preprocessing steps we applied.

The different tools each output a different scale of results. We interpret the results as true or false by setting different thresholds for each tool. The detailed information is shown in Table 1 (Appendix 7.1).

Except for DeepSim, we need to decide the threshold on our own. To account for the effect of the template code given to all students and potential inherent similarity of the project, we calculate the scores between 2 original codes are the baseline similarity score. To distinguish the plagiarized pairs, we randomly picked 10 of the identified true pairs and used the average of these scores as the similarity threshold. By adding the baseline and the similarity threshold, we acquire our final threshold.

Performance Overview

The performances of each tool are evaluated based on the labels generated on the 666 pairs of codes. The visualized results for each tool on each pair of code can be found in Table 2 (Appendix 7.2). We keep the count of the total number of false positive and false negatives and calculate the precision, recall score of each tool using the below formula:

- Precision = (# true positive + # true negative)/ # total points
- Recall = (# true positive) / #positive

The results are summarized in Table 3 (Appendix 7.3). These values serve as a general indicator of how these tools behave, but this is not a very accurate representation of the relative performance between the 4 tools. This is because the total number of positive samples among 666 data points is only 25 which could affect the recall score significantly. To accurately compare these tools, the detailed evaluation for each research question is explained below.

RQ1:How effectively can code similarity tools detect different degrees of code plagiarism?

For this question, we analyze the results from 2 different datasets.

37 Real world data

First is the 37 student submission dataset. Based on manual inspection, we labeled 25 pairs of code as positive. The recall score from the previous section indicates that Gemini, DeepSim, SIM and Plaggie were only able to detect 1,8,1,1 positive cases, respectively.

Even though DeepSim achieved the highest recall rate than the other tools, this does not mean that DeepSim is a superior tool at detecting plagiarism than others. We also noticed that DeepSim has an extremely high false positive count (347) than others. This highest recall score may have been a result from DeepSim abusively labeling data points as positive.

We looked into the one sample that all 4 tools were able to detect: pair (2,19). A side by side comparison is shown in Table 4 (Appendix 7.4). The 2 code segments are almost identical except for a few minor changes. This proves that all tools are at least effective at detecting near identical clones.

Though the results do not seem very promising, this does not mean none of the tools were able to detect similarities in real world data. We suspect this low recall rate may have been caused by the inadequate amount of data and low count of positive samples. Another possible reason could have been the manual labels generated were not accurate enough. The codes we are comparing are relatively big in size compared to data used in original papers for these tools (around size of 1 function). The increase in size and number of functions may have increased the difficulty for plagiarism detection.

Simulated Data

To reduce the problems into smaller scope, we compared these tools on the second dataset: Simulated Plagiarism Set. To represent different types or degrees of plagiarism, we made 4 plagiarized copies of the original work of one team member. The 4 copies each contain:

1. Variable name/type/value change
2. Small code structure change
3. Line addition/removal (print, small computation, new variables/data structures, etc.)
4. Combination of all above changes.

Pairwise comparison between these 4 copies and the original code was conducted and the results are shown in Table 5 (Appendix 7.5). By averaging the performance across all tools, we see a uniform trend that similarity detection tools are more confident at detecting variable changes than structure changes than line removals than combinational changes.

All 4 tools are able to detect plagiarism in the presence of variable changes and structure changes. But for line removal or addition, Plaggie and SIM only reported 50% similarity. All combinational scores are also severely affected by this type of change.

We notice that the two binary tools behave rather consistently across the 4 comparisons, making binary tools rather strong and dependable for purposely obfuscated code. This may be because the binary tools focus more on the functionality of the code rather than syntax. And the minor changes may be attenuated after compiling into binaries.

In conclusion, binary tools (Gemini and DeepSim) are better at detecting different levels of plagiarism than source code and text diff tools. An example of (artificial) plagiarism that binary tools detected but the other tools could not is shown in Table 6 (Appendix 7.6).

RQ2: How do different types of code similarity tools react to original works addressing the same problem?

As shown in Appendix 7.3, the false positive counts for Gemini, DeepSim, SIM and Plaggie are 88, 347, 1, 20 respectively. We notice that DeepSIM and SIM have extremely high or low false positive counts. For DeepSim this may have been the side effect of overly reducing the dimensions of each data point(128*128*88 -> 128), the reduction has attenuated the features of each code and making them all look similar. As for SIM, this low false positive count may be because it is overly conservative at detecting plagiarism. Hence do not reflect a better ability at distinguishing original works. As for the remaining 2 tools, we see that Plaggie is slightly better than Gemini, probably due to it having access to source code.

To give a more detailed example, we run similarity detection using the 4 tools between the team member's original works. The results can be found in Appendix 7.7. SIM and Plaggie both report a fairly low similarity score (~0.4). The binary tools score much larger (0.6 for Gemini and 0.7 for DeepSim). This is likely due to the effect of the given template and the 2 pairs of code have the same functionality. Here we argue that 0.6 for Gemini is an acceptable score, due to the fact that Gemini's scores are typically distributed at a higher average. (So score 0.6 for Gemini and DeepSim is different.)

In conclusion, in distinguishing original codes from each other, SIM and Plaggie are better than Gemini and DeepSim is the worst.

RQ3: Of the approaches examined, which is particularly effective for detecting academic plagiarism in code?

Of the tools considered, the binary code similarity detectors performed the best at detecting plagiarism of different levels, but with a considerable number of false positives. Thus, it is essential to confirm any suspicious submissions caught by these binary code similarity detectors. From our experience, we were often unable to find any resemblance between the works that these tools flagged, but it is difficult to conclude whether they were really false positives or cases of devious plagiarism.

Both the source code similarity and text difference tools were good at catching obvious plagiarism that human eyes would agree without much difficulty. However, the experiment has shown that it is easy to fool these with relatively little effort, namely by adding or removing meaningless, redundant lines of code that introduce new variables/data structures, print statements, and simple if/else logics. On the other hand, Gemini and DeepSim experienced only a marginal drop in the similarity score even when such obfuscation was applied.

Summary:

To summarize, all 4 tools are able to detect plagiarism, but Gemini and DeepSim are better at detecting higher degrees of changes. In distinguishing between original codes, SIM and Plaggie

are better than Gemini and DeepSim is the worst. Overall, binary tools perform well in all aspects and may be more suitable to use in school assignment plagiarism detection settings. Among all types of changes students make to obfuscate, line changes hinders the tools the most.

3. Deliverables

Link to repository: https://github.com/junchun/6156-imposters2

All the materials used in this project is organized in the repository provided above. See the repo's README or Appendix 7.8 for a list of links to important materials.

4. Reuse:

The 37 student submissions were obtained from public repositories forked from: https://github.com/anthonykrivonos/4156-PublicAssignment/network/members. The only submission used in the project that is not listed publicly here is one of our own (Jun) submissions. The copies of these submissions are available in the project GitHub repository at https://github.com/junchun/6156-imposters2/tree/main/data/original_github_data.

The list of tools we used for this project can be found in Appendix 7.9.

While not as extensive, or identical in approach, we referred to [1] as the guideline for our experiment. For instance, we included the tools that [1] tested in our experiment. Unlike this paper, we focused specifically on comparing performances of different types of plagiarism detection tools.

5. Self-Evaluation:

Jun:
I collected original data from GitHub and processed them so they could be used by SIM (text diff tool). I also ran SIM over the processed data. I formulated the research questions and baseline methodology for the required experiments. Also, I worked with Claire to come up with the demo slides and came up with the overall structure and content of the presentation. Working with unprocessed data was a definite challenge, and we kept stumbling upon unusable data: it took us considerable time going over the data alone to check whether they are fit for the experiments (for example, checking whether students actually posted a complete, working solution) and processing them. Throughout the process, I learned much about different types of code similarity detection tools and how they operate--as the topic of this project was new to me (I did automated software testing for my midterm paper), there was a lot to learn. It was also discouraging to learn that students do cheat, but we were glad that our approaches were able to catch at least some possible cases of plagiarism, as well as that we were able to show that tools are effective in catching frequent attempts to plagiarize such as changing variable names.

Claire:
I generated the true labels for the 37 students submissions and was mainly responsible for running the 3 tools: Gemini, DeepSim and Plaggie over the collected data. I analyzed the results of these tools and made figures. I helped with the demo slides and was in charge of gathering needed data.
One thing I found challenging was reusing these open source tools. Most of them only comply with a specific version of python or java or libraries. Without maintenance, such tools are harder to use and I often need to go into their source code and change some stuff for it to run.
Another thing I found challenging was the learning process for binary tools. I also tried training the tools on our gathered data, but due to low count to positive data, the training process is likely to just return a “alway 0” classifier which has low loss but meaningless.
What I have learned in this project is some hands on experience on utilizing open source tools and I proved the rumors that motivated me to do this project -“School uses binary tools to detect plagiarism so cheaters definitely get caught.”. Though I’m not sure about what tools schools use, it definitely proves that binary detect tools are very strong at detecting student level obfuscation. And as a student, if for some reason you want to cheat the programs, adding meaningless global counters and adding and subtracting them might just fool basic source code detectors.

6. Works Cited

[1] Ragkhitwetsagul, C., Krinke, J. & Clark, D. (2018). A comparison of code similarity analysers. Empir Software Eng 23, 2464–2519. [Website]:https://doi.org/10.1007/s10664-017-9564-7

7. Appendix

7.0. Preprocessing procedures for each similarity detection tools

1. DeepSim:
   - DeepSim takes a pair of encodings of the code segment and returns a binary label with probability (confidence)
   - The encodings for each code is 128*128*88 in size and the code used to generate such encodings were given here. However this file requires a deprecated jdk 8 version and a no longer supported WALA version. Therefore, slight modifications were made to its source code and encodings are generated using this project instead.
   - The 128*128*88 sized matrices for each data point (1669 for training and 42 for testing and O(n^2) for number of pairs) were too big to handle using team members' laptops. Therefore dimension reduction was performed to reduce the number of features from 1441792 to 128. The code used to process and reduce dimension is found here.
2. Gemini:
   - Gemini takes a single code segment in the form of a 7-feature-array and returns a graph embedding. Similarity scores are calculated as the cosine similarity based on the embeddings.
   - The original paper mentions the 7-feature array is manually extracted from the binary but did not explain what features were selected. Therefore we used the number of

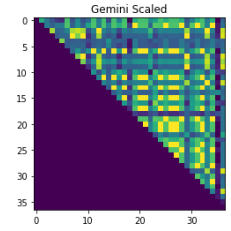
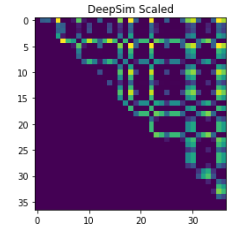
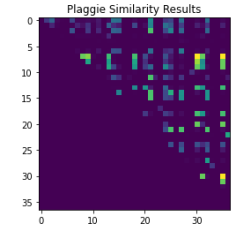
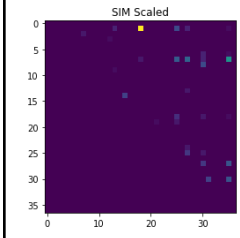
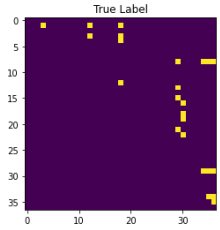
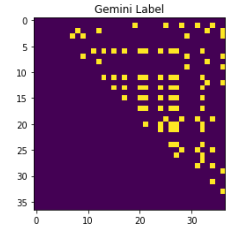
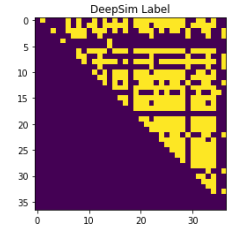
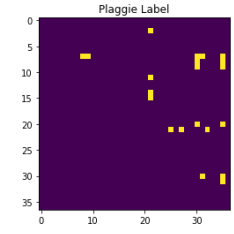
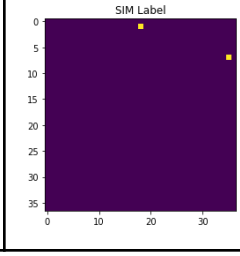
occurrences for 7 different assembly tokens as the feature. The code used to extract features can be found here.

3. Plaggie:
   - Plaggie takes java source code as input and output a percentage score for each pair of codes.
   - Plaggie is using an older version of java parser which does not allow “ctx->{}” notion inside “post” or “get” methods. Since all files have the same endpoints, we removed all “ctx->{}” in all files and this should not affect the similarity results across files.
4. SIM:
   - SIM takes text files as input and output percentage similarity for each pair of codes as well.
   - SIM is unable to compare multiple files in 1 directory with multiple files in another directory. Therefore, we manually merged the 5 java files in each submission into 1

7.1 Table 1: Thresholds for each tool

	DeepSim	Gemini	Plaggie	SIM
Result Scale	0-1 probability	0-1 cosine similarity	0-100 %	0-100%
Threshold	0.5 (Set by tool)	0.79 (0.66 original+ 0.13 similarity)	40% (30.5% original + 11% similarity)	59.8% (47% original + 12.8% similarity)

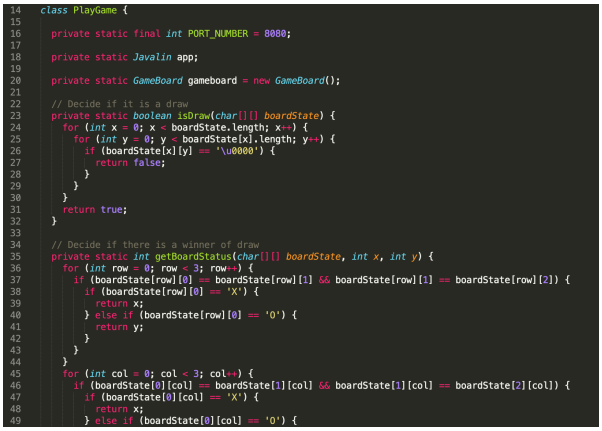
7.2 Table 2: Pairwise comparison results

Gemini	DeepSim	Plaggie	SIM	True Label
				
				

7.3. Table 3: Numerical comparison results

	Gemini	DeepSim	SIM	Plaggie
Precision	83.8%	45.3%	96.3 %	93.54%
Recall	4.16%	33.3%	4.16 %	4.16%
False Positive	88	347	1	20
False Negative	23	17	23	23

7.4. Table 4: Side-by-side comparison of plagiarised works caught



```
class PlayGame {

  private static final int PORT_NUMBER = 8080;

  private static Javalin app;

  private static GameBoard gameboard = new GameBoard();

  // Decide if it is a draw
  private static boolean isDraw(char[][] boardState) {
    for (int x = 0; x < boardState.length; x++) {
      for (int y = 0; y < boardState[x].length; y++) {
        if (boardState[x][y] == '\u0000') {
          return false;
        }
      }
    }
    return true;
  }

  // Decide if there is a winner of draw
  private static int getBoardStatus(char[][] boardState, int x, int y) {
    for (int row = 0; row < 3; row++) {
      if (boardState[row][0] == boardState[row][1] && boardState[row][1] == boardState[row][2]) {
        if (boardState[row][0] == 'X') {
          return x;
        } else if (boardState[row][0] == 'O') {
          return y;
        }
      }
    }
    for (int col = 0; col < 3; col++) {
      if (boardState[0][col] == boardState[1][col] && boardState[1][col] == boardState[2][col]) {
        if (boardState[0][col] == 'X') {
          return x;
        } else if (boardState[0][col] == 'O') {
```

```
class PlayGame {

  private static final int PORT_NUMBER = 8080;

  private static Javalin app;

  private static GameBoard board;

  //Lets us know if a board shows a draw
  private static boolean isDraw(char[][] boardState) {
    for (int x = 0; x < boardState.length; x++) {
      for (int y = 0; y < boardState[x].length; y++) {
        if (boardState[x][y] == '\u0000') {
          return false;
        }
      }
    }
    return true;
  }

  //Goes over the rows and then the columns, and finally, diagonals, to determine possible winner
  // or draw
  private static int getBoardStatus(char[][] boardState, int x, int y) {
    for (int row = 0; row < 3; row++) {
      if (boardState[row][0] == boardState[row][1] && boardState[row][1] == boardState[row][2]) {
        if (boardState[row][0] == 'X') {
          return x;
        } else if (boardState[row][0] == 'O') {
          return y;
        }
      }
    }
    for (int col = 0; col < 3; col++) {
      if (boardState[0][col] == boardState[1][col] && boardState[1][col] == boardState[2][col]) {
        if (boardState[0][col] == 'X') {
          return x;
        } else if (boardState[0][col] == 'O') {
          return y;
        }
      }
    }
```

Code 2	Code 19

7.5. Table 5: Pairwise comparison of detecting artificial plagiarism

	Lines added and removed	Variables changed	Structures changed	All changed
GEMINI	0.877	0.972	0.999	0.8948
DeepSim	0.8909	0.9546	0.8696	0.8354
Plaggie	50.2%	77.9%	68.8%	46.1%
SIM	49%	79%	92%	41%

7.6. Table 6: Side-by-side comparison of artificially plagiarised works

<table>
<tr>
<td>

```java
/**
 * Check if the move is valid or not.
 * If invalid, a message appears to the players.
 * @param move player move
 * @return false if the move is invalid
 */
public boolean isMoveValid(Move move) {
  char[][] board = getBoardState();
  if (getTurn() != move.getPlayer().getId()
      || board[move.getMoveX()][move.getMoveY()] != '0') {
    return false;
  }
  return true;
}

/**
 * Set a valid move and update overall game status.
 * Should be used after isMoveValid() method to check if the move is valid.
 * @param move move
 * @return true if the game is over
 */
public boolean setMove(Move move) {
  char[][] board = getBoardState();
  board[move.getMoveX()][move.getMoveY()] = move.getPlayer().getType();
  if (getTurn() == 1) {
    setTurn(2);
  } else {
    setTurn(1);
  }
  return checkOverallStatus(board, move);
}
```

</td>
<td>

```java
public int ssmmm113(Move move) {
  int three = 3;
  int four = 4;
  System.out.println(three + four);
  char[][] board = ggeeee();
  board[move.whatsTheNextMoveX()][move.whatsTheNextMoveY()] = move.getHimHere().retrievePlayerType();
  if (iiiee() == 1) {
    ssttt(2);
  } else {
    ssttt(1);
  }
  return ccooss444(board, move);
}

public int makeIt2nd(Player bb) {
  int three = 3;
  int four = 4;
  System.out.println(three + four);
  this.p2 = bb;
  return 0;
}
```

</td>
</tr>
<tr>
<td>original</td>
<td>all_changed</td>
</tr>
</table>

7.7. Table 7: Pairwise comparison of detecting original code

Tool	Similarity score
DeepSim	0.73
GEMINI	0.656
Plaggie	30.5%
SIM	47%

7.8. Links to Key Project Materials

All Project Assignments
https://github.com/junchun/6156-imposters2/tree/main/all_submissions

Data Used for This Project
Original Student Submissions:
https://github.com/junchun/6156-imposters2/tree/main/data/original_github_data
Artificial Plagiarism data:
https://github.com/junchun/6156-imposters2/tree/main/data/simulated_plagiarism
Processed Data for Plaggie:
https://github.com/junchun/6156-imposters2/tree/main/tools/plaggie/plaggie_data
Processed Encoding Data for DeepSim:
https://github.com/junchun/6156-imposters2/tree/main/data/encoding_data

Code, Configurations:

Gemini Code:
https://github.com/junchun/6156-imposters2/tree/main/tools/gemini
Plaggie Code:
https://github.com/junchun/6156-imposters2/tree/main/tools/plaggie
Plaggie Configuration:
https://github.com/junchun/6156-imposters2/blob/main/tools/plaggie/plaggie.properties
DeepSim Code:
https://github.com/junchun/6156-imposters2/tree/main/tools/deepsim
DeepSim Code used to generate Encodings:
https://github.com/junchun/6156-imposters2/tree/main/tools/generate_encoding
SIM Code:
https://github.com/junchun/6156-imposters2/tree/main/tools/sim-master
(originally from: https://github.com/andre-wojtowicz/sim)

7.9. List of Tools Used (with Original Sources)

Gemini: https://github.com/xiaojunxu/dnn-binary-code-similarity
DeepSim: https://github.com/parasol-aser/deepsim
Plaggie: https://github.com/mdaum/PlayingwPlaggie
SIM (original): https://dickgrune.com/Programs/similarity_tester/
SIM (working version used for this project): https://github.com/andre-wojtowicz/sim