PLAGIARISM DETECTOR REPORT

TEAM 21

VISHRUTH K

ANKITA PATEL

CHAITANYA KAUL

ROSY PARMAR

Abstract

The goal of this project is to design, implement, test, and evaluate an application to help instructors detect situations where two or more students submit similar solutions to an assignment. Examples of such transformations are- moving functions or methods to another location in the same file, renaming variables, classes, and methods, extracting sequences of statements into methods, etc.

High Level Design Overview

- User logs into the system and uploads two java files to check for plagiarism.
- Once the files are uploaded, AST factory creates two ASTs of the uploaded files.
- When ASTs are constructed, the Comparator Factory is triggered which then implements IComparator and its respective methods overridden in its child classes. The results for similarity percentage between two input files are calculated in these child classes based on line count, comments similarity and structural similarity.

We have implemented Factory Method Design Pattern for the system. We have two factories namely, ASTFactory and CompareFactory. The two factories are created such that the implementation of AST creation and Tree comparison are hidden from the client who intends to use the system.

Algorithm

The algorithm can be divided into three parts:-

1. LineCount Comparison
2. Comment Similarity Comparison
3. Structural Similarity

Line Count Comparison

Line Count Comparison is very basic in terms of implementation. We used simple while loops for counting the number of lines in the two files. After that, we keep a threshold of around +10/-10 lines for matching the count for both the files.

Comment Similarity Comparison

For Comment Similarity, we use JavaParser's CompilationUnit object which points to the root node of the Abstract Syntax tree for a given JAVA file. We extract all the single line comments as well as multiline comments from both the JAVA files and then remove all the stop words (common English words like a, an, the etc.) from both the lists that we get as output.

Finally, we sort all the words that we now have and apply Levenshtein's algorithm on both the lists and get the similarity distance and finally after normalizing, we output the percentage.

Structural Similarity

For finding the structural similarity we have used one of the well-known methods (Hashcode Comparison). The algorithm can be divided to separate parts:

- The first part involves generation of Abstract Syntax Tree for both the JAVA files.
- AST's generated are now traversed and the hashcode generation algorithm is called and different hashcodes are assigned to each of the nodes. The hashcode is derived from the nodeType() method of JDT parser.
- Now, we generate two hashmaps which contain (key, value) pairs in which the key is a typical ASTNode and the value is our own custom data type containing the hashcode, number of child nodes, node type, line numbers and a boolean identifying if the particular ASTNode is visited or not.
- Both, the hashmaps are now sorted and then iterated upon and then the individual hash values are compared for two nodes at a time from both the hashmaps. If the hash values are same, the node is marked as visited and the parent's hash value are now checked for these nodes. If the hash value of the parent nodes is also same, that signifies that the whole block is copied.
- We export all the resultant blocks to text files which is read by the frontend, that uses the line numbers to show the lines which are copied in the suspected JAVA file.

Approach Implemented

The system was developed in Test Driven Development manner. The team thoroughly explored and built all the possible test cases that the system would work on. After implementing the functionality, more test cases were added to get a complete test coverage for the entire code. The system successfully predicts the following scenarios:

- If two given input files have same number of line count.
- If two given input files have similar code but structurally transformed.
- If two input files have similar comments.

Furthermore, for the user to use the system with ease, UI is kept very simple. The user after submitting the two files, is easily able to find similar contents in the suspected file marked in green color. To make a fair judgement, the user can view the statistics depicting the similarity percentage between two files based on three parts described in algorithm.

Changes Incorporated

- In Phase B, the return type for all the child classes of Comparator class were String. We have now changed the return type to double. For the line count similarity, if line count is not similar, we return -1, otherwise, we return 0. For similarity returned in other

classes, we return the double value of the percentage. This led to significant refactoring throughout the code since now we do not handle string to double conversion in the code.

- The team has worked on refactoring the existing code into small modular functions to keep the code more readable, understandable, testable and maintainable. Having the tests written at first, helped us in ensuring that the refactoring did not produce any defects in the system.
- The team has also incorporated documentation in JavaDoc style as suggested in the review to ensure that the code is easily understandable.

Experiments Conducted

The team spent a great deal of time experimenting, by passing different code snippets to the system. These included using the following scenarios.

- One of the files having entire code in a single line and the other file having entire code in java formatted style.
  For the above scenario, tool was able to detect plagiarism giving results with extremely high percentage.
- Both the files having exactly similar code.
  The tool was able to detect 100% plagiarism in the above scenario.
- Uploading two empty files.
  The tool accepts the input files and return 0% which is the expected output.
- Uploading files of other format
  The tool does not accept any file which does not have a .java extension and displays a error message on the UI.

References:

[1] Zhao, J., Xia, K., Fu, Y. and Cui, B., 2015, November. An AST-based code plagiarism detection algorithm. In Broadband and Wireless Computing, Communication and Applications (BWCCA), 2015 10th International Conference on (pp. 178-182). IEEE.

[2] Levenshtein Distance, https://xlinux.nist.gov/dads/HTML/Levenshtein.html