Abhishek Mulay, Daniel Daskivich, Piyush Goel, Robin Nag
CS 5500 – Section 2, Professor Michael Weintraub
Final Report

Introduction

The goal of this project was to design, build, and test a program that will aid instructors to detect plagiarism and academic dishonesty. This program focused on Java programming assignments specifically. This paper explains the system functionality, design overview, the plagiarism detector algorithm, experiments, performance, techniques, and the technology stack.

System Functionality

The program is a web application that can be accessed over the internet. It is only accessible by professors, who have valid login credentials. Once logged in, the user drags and drops a zip file which contains all submission directories as subdirectories. Once the submission file has been uploaded, the user clicks on the "Detect Plagiarism" button to get the results. The results page will output submission pairs in descending order of match score (1 being a complete match and 0 being no match). Each submission pair contains the following data: the file path for each submission, overall match score, the individual factor scores, the number of matches found for each factor, and an option to see more details. The additional details show the actual matches for each factor and the line numbers of where the matches occurred. Once the user has seen the results, he/she can log out of the web application. None of the data will be stored to assure protection.

Design Overview

The system was designed using the model-view-controller (MVC) design pattern. Methods are declared in the SubmissionComparatorInterface and are implemented by SubmissionComparator (the model). The RestController (the controller) contains the functionality to perform the submission comparisons requested by the view and access to the resulting comparison data that will be displayed by the view. Within the model, the third-party JavaParser library is used to create an abstract syntax tree (AST) for each ".java" file from each submission. The model then uses multiple strategies to pull data from each submission's ASTs. This data is needed to perform the actual code comparisons. This preprocessing step is performed only once per submission and tracked by the model as a list of PreprocessedSubmission objects. Some preprocessing strategies utilize the visitor pattern implemented by JavaParser; others use existing JavaParser convenience methods. Each strategy identifies a unique data set from each submission that will be compared with other data sets in a unique way. Various actions may be taken by plagiarizers to help obscure the fact that they have

plagiarized. The hope with our design is that considering similarity from multiple perspectives will help uncover this underlying connection. Finally, a Comparison object is created for each unique pairing of PreprocessedSubmission objects. Each Comparison uses four Factor classes, one for each matching strategy, to perform the actual comparison of data sets created during the preprocessing step. The resulting details of each comparison are stored in a ComparisonDetails object, which is tracked by the model as a list of ComparisonDetails objects and ultimately passed to RestController.

Algorithm

Comparison results created by the system represent the relative likelihood that one of the submissions from a particular comparison contains plagiarized code; the higher the result, the higher the likelihood of plagiarism. Such results are most useful when analyzing a multitude of submissions— for example, when uploading submissions for an entire class. Each submission is compared with each other submission, and the resulting pairwise comparison results are displayed in descending order of plagiarism likelihood. It is recommended that the system user works his or her way down the list, manually compare source code matches for comparisons from the top of the list first, until plagiarism is deemed unlikely by human judgement. The comparison result for a particular comparison is actually the average of several component "factor" scores, where each "factor" employs a different strategy for comparing source code. If a particular matching strategy is irrelevant for a particular comparison, that strategy's factor score is not considered when calculating the final matching result; e.g., if one of the submissions contains no comments, then the comment matching strategy is considered to be irrelevant for that comparison.

Presently, the system employs four matching strategies: identifier matching, comment matching, literal matching, and statement matching. For identifier matching, the system compiles a list of names used for classes, interfaces, methods, and variables in each submission; these lists are then compared, and the reported identifier factor score is the percentage of identifiers from the smaller list that also appear in the larger list. For comment matching, the system compiles a list of the contents of comments from each submission, extracting these contents from each type of comment (block comments, javadoc comments, and line comments); these lists of comment contents are compared, and the comment factor score is the percentage of comment contents from the smaller list that also appear in the larger list. For literal matching, the system compiles a list of the literal values from each submission (booleans, chars, doubles, ints, null, longs, Strings); these lists of literals are compared, and the literal factor score is the percentage of literals from the smaller list that also appear in the larger list. Finally, for statement matching,, the system compiles a list of statement blocks from each submission. For each statement block, the system counts the number of each statement type found within that block: assert statement, block statement, break statement, continue statement, do statement, empty statement, explicit constructor invocation statement, expression statement, for each statement, for statement, if statement, labeled statement, local class declaration statement, return statement, switch entry statement, switch statement, synchronized statement, throw statement, unparsable statement, and while statement. If two statement blocks contain the same count for each type of statement, these

two statement blocks are considered to be a match. The statement factor score is the percentage of total statements from the submission with fewer statements that are contained within statement blocks that match statement blocks from the other submission.

Experiments/Performance on Sample Test Data

In order to test the effectiveness of the algorithm, we utilized sample test data from both the instructor and ourselves. The instructor sample test data contained a total of 20 sets of test data files. Each set had two submissions. Some of the sets contained very similar submissions, whereas others contained completely different submissions. We had to manually look over the files to check if our algorithm was detecting the factors that we wanted. Our algorithm was very effective in detecting matches for identifiers, comments, literals, and block statements. For example, Set 9 contained two very different submissions. As a result, its overall score was 0.15. On the other hand, set 11 contained almost the same submissions. Based on our factors and their properties, its overall score was 1.0. Based on these experimental runs, we tweaked our algorithm to detect more subtle matches in our four factors. Even though more factors can be considered for matching, our current algorithm is able to detect intermediate to major cases of plagiarism.

Technique

In addition to the design patterns referenced above, we utilized black-box (unit tests) and white-box (statement coverage) testing strategies, and built a backend test suite for regression testing. We refactored backend design to accommodate separate functionality for preprocessing in an effort to improve algorithmic efficiency by eliminating preprocessing redundancy. We refactored code structure for various methods to increase code readability, composing smaller methods out of larger ones, and creating private helper methods.

In terms of design patterns, we used the visitor and the strategy patterns. The JavaParser library built an AST for each Java file. In order to traverse the AST, we used the visitor design pattern. In addition, we used the strategy pattern to decide what kind of data to extract from an AST so that we can perform the comparisons.

Technology

We used a mixed technology approach to create our program. For backend development, we used a Spring Boot application with the Spring MVC framework. This framework provided controllers, http request mapping, and conversion of Java objects to JSON, which were all important for communication with the front-end. Java was used as the programming language, JUnit was used for testing, and Maven was used for dependency management. The following important dependencies were used: JavaParser, Guava, Common IO, and JSON.simple. For front-end development, AngularJS was utilized. Finally, Git was used for version control.

Future Work

Possible future work could include the display of source text for a selected comparison, the implementation of additional comparison strategies, the implementation of reporting functionality to create an exportable document outlining the findings of a particular comparison, and the use of a longest common subsequence algorithm for identifier/comment matching rather than strict equality.