CodeChecker - Plagiarism Detection Tool

by

Kshitiz Adlakha, Eric Rollins, Preethi Chavely, Mahesh Kumar Bhaskaran

System Functionality

CodeChecker is designed to compare the similarity between two Java files via a web interface. Once a user has created an account in our system, he is prompted to login. Upon logging in, the user is prompted to type a project name and to upload two java files to compare for plagiarism. After submitting this comparison, the user will be brought to a side by side view of the two java files that displays the percent of plagiarism detected. In this view, the user can toggle between viewing the original files and viewing the transformed files. The user also has the option of viewing previous comparisons.

When the users submit java files for comparison, both files go through a numbers of manipulations in order to be properly compared against each other. The files start by being parsed using the JavaParser library. This functionality allows CodeChecker to convert the two submitted files into abstract syntax trees. This conversion allows the algorithm, described in more detail below, to perform a series of standard manipulations. The two standardized java files are then run through a comparison calculation (described below). The results of this calculation and transformations are then returned to the end user for review.

Design Overview

CodeChecker is divided into two primary components: a client-side application, implemented using Angular, HTML, and CSS, that lets user register on the application and upload programming assignments that need to be checked for plagiarism, and a server-side application, implemented using Spring MVC, and Java Persistence API, that performs the plagiarism detection using a complex algorithm, explained in detail below, and returns a similarity percentage between the two assignments. As per our initial report, we had planned to restrict the access of the application by adding an ‘administrator’ user, who would approve each sign up request, but for the current version of the application, we have removed this part and anyone can access the application. We also found that the ‘Course’ entity that we had listed in our initial UML diagrams was redundant, and has been removed in the current version.

Algorithm Used

Once the codes are submitted, they are converted to Abstract Syntax Trees (AST) using JavaParser. They undergo preprocessing in order to standardize the codes in terms of function names, comments, variable names, data types, class names, etc. Once the preprocessing is done, we standardize the order of the methods, variables, and classes present in the java file. After all this is done, we get transformed files, on which we perform a similarity percent calculation.

1. **Preprocessing**
    a. Comment removal.
    b. Datatype standardization. All datatypes are converted to type "T"
    c. Variable, method, class and packages name standardization: Replace identifier, method, class, and package names with standardized names.
2. **Reordering**

    After preprocessing is done, the order in which the methods appear in the java file are ordered based on the number of child nodes, number of parameters the method contains, and the data type returned. Next, the variables are sorted on the basis of their types and given standard names in the order they are encountered.
3. **Similarity Percent Calculation**
    a. Visit each node in the AST
    b. Obtain the hashcode of each node and store in a HashSet

    *Repeat steps **a** and **b** for the other AST as well.*
    c. Take the intersection of the two HashSets to get the nodes that are common in both ASTs and store it in a HashSet S.
    d. Remove the hash codes present in S from the first HashSet and store the remaining unique hash codes in a HashSet called L.
    e. Remove the hash codes present in S from the second HashSet and store the remaining unique hash codes in a HashSet called R.
    f. Calculate the similarity percent using the formula adapted from Clone Detection Using Abstract Syntax Tree[1]:

    $$Similarity\ \% = ((2 * S.size())/((2 * S.size()) + L.size() + R.size())) * 100$$

Techniques covered in this course

Test driven development was incorporated in our plagiarism detection module by writing tests before writing functional code followed by refactoring. Refactoring of the code was done by renaming variable names, function names and documentation. Refactoring helped improve code readability and simplify code structure. We used the visitor design pattern to visit the nodes in AST, and to transform the AST generated, as well as to add external classes to our plagiarism detector without modifying other classes. MVC pattern was used for implementing the user interface, as well as the server-side application. We used branch and statement coverage to ensure all our branches and statements were executed in our plagiarism detector module.

Sample Projects and Experimentation

We ran the Sample Project containing 20 sets against Code Checker. Some results are recorded here. Similarity percent for the first set was 78.01%, this would indicate that there is a high chance that these two files were plagiarized. Similarity percent for the second set was 39.36%, this would indicate that there is a low chance that these two files were plagiarized. Similarity for the sixth set was 92.06%, indicating a very high probability of plagiarism. Set 12 throws an error as the files do not have java extension.

In addition to the sample projects, we also experimented with empty files, non-java files, duplicate files, and distinct files. These edge cases allowed us to thoroughly test our code and guard against unexpected errors.

Through manual observation, and the looking at the transformed files, we came to the conclusion that CodeChecker is able to identify the files that were plagiarized with good accuracy. The transformation of the files to a standardized form, helped in the elimination of false negatives.

Limitations and Enhancements

Currently our system has a few limitations. CodeChecker can only compare individual files. In the future, this should change to compare projects against one another. On top of that, variable and function renaming is currently done without respect of scope. In the future, object and function scope should be considered when renaming.

References:

1. *Clone Detection Using Abstract Syntax Trees* by Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, Lorraine Bier.
2. *Detection of plagiarism in Java programming assignments in introductory computer science courses at Ifi* by Jose Luis Rojas