Duplicate Code Detection System

Arun Cheriakara
Department of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1, Canada
acheriak@uwaterloo.ca

1 Introduction

Duplicate code, also known as code duplication or cloning, is a critical problem in software development. It occurs when developers replicate existing code instead of reusing it through modular functions or classes. While it may seem harmless at first, duplicated code can lead to increased maintenance efforts, bugs, and technical debt as projects evolve.

The following paper is a proposal for a Duplicate Code Detection System which is supposed to help in identifying and grouping code clones in a particular directory. The system classifies duplicates into three categories namely; exact duplicates, near duplicates and structural duplicates. Employing sophisticated approaches to string similarity, Abstract Syntax Tree (AST), and hashing, the system tries to enhance the quality of the code as well as the productivity of the developer

2 Motivation

The rise of modern software engineering practices has highlighted the need for automated tools that detect and eliminate duplicate code. Duplicate code not only inflates the size of the codebase but also makes debugging and refactoring more error-prone.

The primary motivation behind this project was to create a lightweight, customizable tool that developers can run locally to detect duplicates in a variety of formats, including .js, .ts, .jsx, and .tsx. Additionally, the tool should allow users to categorize duplicates and save results for further analysis.

3 Project Details

3.1 Objective

The main objective of the project was to develop a command-line tool that:

1. Scans a specified directory for JavaScript and TypeScript files.
2. Identifies three types of code duplicates: exact, near, and structural.
3. Provides a user-friendly interface to choose the type of detection and optionally save the results to text files.

3.2 Implementation

3.2.1 File Scanning and Validation

The tool begins by prompting the user to input a directory path. Using file scanning utilities, it recursively searches for all files with supported extensions **(.js, .ts, .jsx, .tsx)**. The **validateDirectory** function ensures the directory exists and is accessible, reducing runtime errors.

3.2.2 Duplicate Detection

The tool categorizes duplicates into the following types:

- **Exact Duplicates**: Identifies identical blocks of code using hashing algorithms (e.g., MD5). This is computationally efficient and provides an immediate match for identical code segments.
- **Near Duplicates**: Compares code similarity using the string-similarity library. A normalization step removes whitespace and comments before comparison to ensure accuracy. The similarity threshold (default: 0.8) can be adjusted to control sensitivity
- **Structural Duplicates**: Parses the code into an Abstract Syntax Tree (AST) and analyzes the structural logic. This enables detection of logically similar but syntactically different code.

3.2.3 Exact Duplicates

Exact duplicates are identical code blocks replicated across files or functions. These are often the easiest to detect and correct, as no analysis of syntax or logic is required.

Detection Method

- A hashing algorithm, such as MD5, is used to compute a hash for each function. Identical code blocks will generate the same hash value.
- The system compares hashes instead of the actual code, making the process highly efficient.

The hash-based detection ensures that exact duplicates are identified in constant time per function, making it highly scalable.

Example for exact duplicate

```
#file1.js and file5.js

   function calculateRectangleArea(length, width) {
     return width * length; // Reordered but logically identical
   }

The exact duplicate will print out:

   {
     "function": "function␣calculateRectangleArea(length,␣width)␣{\n␣␣
         return␣width␣*␣length;\n}",
     "files": [
       "test\\TestingFilesEndingWithJS\\file2.js",
       "test\\TestingFilesEndingWithJS\\file5.js"
     ]
   }
```

Figure 1: Exact Duplicate Example

3.2.4 Near Duplicates

Near duplicates are code segments that are logically identical or highly similar, but with slight differences in syntax or structure. These differences might include:

- Variable renaming.
- Minor formatting changes.
- Rearrangement of statements without altering functionality.

Detection Method

1. **Normalization**: The tool preprocesses the code by:
   - Removing comments.
   - Normalizing whitespace.
   - Standardizing variable names where possible.
2. **String Similarity**: The normalized code is compared using a similarity algorithm (e.g., string-similarity). A similarity score is computed, and matches exceeding a predefined threshold (default: 0.8) are considered near duplicates.

```
   #File 6
   function calculateRectangleArea(length, width) {
     if (length <= 0 || width <= 0) {
       console.error("Invalid␣dimensions!");
       return 0;
     }
     return length * width;
   }

   #File 7
   function calculateRectangleArea(length, width) {
     if (length > 0 && width > 0) {
       return length * width;
     }
     throw new Error("Invalid␣dimensions!");
   }

The near duplicate will print out with a similarity of 0.8203125:

   {
     "function1": "function␣calculateRectangleArea(length,␣width)␣{␣if␣(
         length␣<=␣0␣||␣width␣<=␣0)␣{␣console.error(\"Invalid␣dimensions
         !\");␣return␣0;␣}␣return␣length␣*␣width;␣}",
     "function2": "function␣calculateRectangleArea(length,␣width)␣{␣if␣(
         length␣>␣0␣&&␣width␣>␣0)␣{␣return␣length␣*␣width;␣}␣throw␣new␣
         Error(\"Invalid␣dimensions!\");␣}",
     "similarity": 0.8203125,
     "files": [
       "test\\TestingFilesEndingWithJS\\file6.js",
       "test\\TestingFilesEndingWithJS\\file7.js"
     ]
   }
```

Figure 2: Near Duplicate Example

3.2.5 Structural Duplicates

Structural duplicates represent logically similar code blocks that may have completely different syntax. This is the most complex type of duplicate to detect, as it requires an understanding of the code's structure and semantics.

Detection Method

1. **Abstract Syntax Tree (AST) Parsing**: The code is converted into an AST, which represents the structure of the code
2. **Structural Comparison**: The tool compares AST nodes to identify similarity in logic. Even if the code is written differently, identical AST patterns indicate structural duplication.

```
The structural duplicate will normalize the function and print the output

  {
    "normalizedFunction": "function␣placeholder(placeholder,␣placeholder)␣
        {\n␣␣return␣placeholder␣-␣placeholder;\n}",
    "originalFunctions": [
      {
        "file": "test\\TestingFilesEndingWithJS\\file4.js",
        "function": "function␣subtract(a,␣b)␣{\n␣␣return␣a␣-␣b;\n}"
      },
      {
        "file": "test\\TestingFilesEndingWithJS\\file4.js",
        "function": "function␣add(a,␣b)␣{\n␣␣return␣a␣-␣b;\n}"
      }
    ]
   }
```

Figure 3: Structural Duplicate Example

3.2.6 User Interaction

The user interacts with the tool via prompts powered by the **inquirer** library:

1. Input the directory path to scan.
2. Choose the type of duplicate detection.
3. Decide whether to save the results to text files **(exact.txt, near.txt, structural.txt)**.

3.2.7 Saving Results

The results of each detection type are saved in a structured JSON format in a text file. This enables easy integration with other tools for further analysis or visualization.

4 Results and Analysis

The tool was tested on a set of sample JavaScript and TypeScript files containing varying levels of duplication. The following observations were made:

1. **Exact Duplicates**: Accurately detected identical functions across files.
2. **Near Duplicates**: Effectively identified minor variations in code structure (e.g., variable renaming or formatting changes)
3. **Structural Duplicates**: Detected logical similarities, such as identical loops written with different syntax.

4.1 Performance

- For smaller datasets (<100 functions), results were instantaneous.
- For larger datasets, optimizations like hashing and normalization significantly reduced runtime, but further improvements are needed for structural comparisons on larger codebases.

5 Future Improvements

1. **Parallel Processing**: Utilize worker threads to handle pairwise comparisons for larger datasets.
2. **Enhanced Structural Analysis**: Implement more advanced AST analysis techniques to improve precision.
3. **Customization**: Provide APIs or CLI options to integrate the tool with CI/CD pipelines for continuous monitoring of code duplication.
4. **Extend for other code bases**: Support applications like python, C and so on.
5. **Extension**: Add a vs code so users can download and test it on their files.

6 Conclusion

This project shows how a tool can be developed to have a high degree of flexibility and efficiency as a code duplicate detection tool focused on JavaScript and TypeScript projects. With the integration of a good CLI interface, the application of detection algorithms can be made easy while at the same time being very powerful. There are however some problems that affect the scalability and the level of accuracy that the system can achieve, however the groundwork provided in this paper is a solid base for development. Detection of code duplication at the initial stage is very beneficial for the quality of software, its further support, and the efficiency of the developers.

References

https://github.com/platisd/duplicate-code-detection-tool

https://www.npmjs.com/package/inquirer

https://www.npmjs.com/search?q=string20similarity

https://ts-ast-viewer.com/