Lex and the Winnowing Algorithm

Tokenized Plagiarism Detection

Parker Corbitt

08.03.2024

REGULAR EXPRESSIONS

My decision on how to tokenize resulted from one line of thought.

1. The projects are inherently basic, meaning that most of what students can change to avoid plagiarism are:
    a. Data Types - Changing an int to a long long, or a double.
    b. Relational Operators - Making equivalent expressions using different relational operators (ex. if(x == y) & if(!x == !y)).
    c. Values and Literals
    d. Loop Types - Changing a for() to a while() loop
    e. Function Names

Minding the fact that the projects are limited in their ability to be unique, I also accounted for a few built-in pieces of functionality by tokenizing specific common functions (ex. main, printf, scanf), as these can be seen as key “checkpoints” within basic programs such as these.

As for the layout of the tokens, covering as generally as possible seemed the best approach. For example, students may try to change an int to a double in order to seem like they are being more creative. However, if the statements “int x = 5” and “double y = 5.0” both convert to NUMBER VAR ASSIGNMENT VAL, the possible obfuscation is lost. A similar story follows for arithmetic, logical, and relational operators, as it is easy to build equivalent expressions by utilizing different operators.

The tokenization of common functions (main, printf, scanf, conditionals, and loops) is also implemented in order to limit the amount of “unique” FUNC tokens.

The changing of literals and general values also tends to be somewhat simple. They are tokenized generally to reflect this.

WINNOWING

The process of implementing the Winnowing algorithm was broken up into 7 steps

1. Read the input tokens and remove any unnecessary text (whitespace mainly)
2. Split into k-mers
3. Hash the k-mers
4. Window the token’s respective k-mers into their respective w-windows
5. Select the minimum hash from each window
6. Compare the list of minimum hashes of each file to the remaining files to be compared against
7. Score the probability of plagiarism based on the comparison
8. Order and print the scores if they are above t - threshold

ANALYSIS

Generally all of the files were somewhat similar. This was expected due to the limitations of uniqueness in such a simple program paired with my method of tokenization. There were some notable results though.

With a k-value of 8, My program determined that the programs numbered 3, 9, 53, along with 15, 47, and 49 were 100% plagiarized. Upon further inspection of these files, those results are verifiable. The structure is the same, with the exact same logic. Even some of the literals are verbatim.

Notably, it goes on to mention that 10 and 30 are ~99% similar. When reading those files, another note to make is that they look nearly exactly the same as the previous 4. Running the program with a smaller k-value (4) yields 10 and 3 having a similarity rating of ~94%. Inspecting the file yields some functions being different, but large sections being plagiarized. Not total plagiarism, but clearly not well hidden.

These two instances of results communicate that this program isn’t well suited to just a single run with only one k-value. Should you only try once, you may not get the entire picture. It is highly recommended to run multiple times, and analyze the high probability files, tuning the k-value as you proceed.