CS 3110 Final Project - OCaMOSS

Aniroodh Ravikumar (ar765), Danny Yang (dzy4), Gauri Jain (gj82), Robert Yang (ry92)

System Description

Idea: Plagiarism Checker - create a program that detects similarities between files, checks for any signs of plagiarism and judges the severity of plagiarism between the files uploaded. Our idea is based on the MOSS system[1].

Key features:

- Compares structural/syntactic similarity across files of programs written in the same language and text files
- Provides a similarity score between the given files, giving a rough idea of what score ranges correspond to likelihood of plagiarism
- Displays detailed view of overlaps/matching patterns of code between two files
- The program works for OCaml, Java, Python, C, and text files and can be extended to other languages quite easily

Description:
This program would allow the user to check the similarity of multiple code-based files of the same file type. The user will provide the files in one specified directory, and run the program which would operate on all the files in that directory. Our plagiarism checker would then display a score of similarity score and highlight which portions of the files are similar, allowing the user to then go through those parts by themselves in more detail. The plagiarism checker is whitespace and position insensitive, in addition to being insensitive to variable/function name manipulation and changes in hardcoded strings and comments.

While the program is be able to detect plagiarism between files of a single filetype, we will be able to provide support for multiple languages. The addition of support for a new language is possible by adding a json configuration file containing keywords specific to that language to the main directory of the program.

Changes

Support for new languages - We added support for Java, C, Python and regular txt files in addition to OCaml. Text files are only checked for direct copy-pasting and do not have any synonym checking functionality.

[1] http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf

Pretty Printing - Results are displayed in a two-column format in the terminal for easy viewing. Some printouts are colored, with error messages being red and completion messages being green.

Similarity Scores - Initially, we had only displayed the list of files that we found to be plagiarised. We now display a similarity score between each pair of files, as well as an overall similarity score for each file that's the average of it's similarity score with files that it has been found to be plagiarised with.

Preprocessing - We remove comments and strings from the file before hashing. If applicable, we support both single and multi-line comments, as well as nested comments.

Testing

Preprocessing - All of the preprocessing functions are individually tested in the test suite. The functions are tested mainly for edge cases, and smaller inputs. Testing still needs to be carried out for files of larger inputs.

Winnowing - Tests for the winnowing algorithm were generated in two ways. The simpler tests were calculated by hand, while longer tests with larger window sizes and up to 100 input hashes were generated using a Python implementation of the same algorithm using random numbers for hashes and manually reviewed for correctness. The OUnit tests check a string representation of the output. All test cases for winnowing pass, and since some were randomly generated we are relatively confident that the algorithm works as intended.

Dictionary - The main tests written to check dictionary were a few member/find tests on a very large tree with various two and three nodes, as well as a large insert operation checker converting the entire tree to a list. We also completed very thorough utop testing to make sure the structure of the tree was correct throughout the insert operations. We used the insert example from the referenced Wellesley College worksheet[2] to make sure our insert logic was correct.

Comparison - Tests for comparison at this point have been tests for the individual functions through the test suite. Most of the tests have been testing edge cases, and test cases are generated by hand since unlike winnowing, testing of comparison does not particularly require testing with huge cases.

End to end tests -

- The metrics used to measure the effectiveness of our program are precision, recall and F1 score, which are the standard metrics for document classification tools.

[2] http://cs.wellesley.edu/~cs230/fall04/2-3-trees.pdf

    - Precision is the percentages of files we marked as plagiarised that are actually labelled as plagiarised (A low precision means that we have a lot of false positives).
    - Recall is the percentage of files we marked as plagiarised amongst the files we should have found to be plagiarised(A low recall means that we have a lot of false negatives).
    - F1 score is the harmonic mean of precision and recall that gives a good effectiveness score by taking into account both precision and recall.
- In all of the tables below, we're only going to consider pairwise comparisons as a reliable metric since it is possible to flag a file as plagiarised, and not catch all of the files that it plagiarises from/with.
- While in the latter table, we were provided with a labelled dataset, in the former, we were able to measure accuracy ourselves since they were self-generated test cases, and in the vast majority of the cases, blatantly plagiarised with minor changes.
- Our overall results for the effectiveness of OCaMOSS are calculated using a threshold of 0.4, which was experimentally determined to be the best. Increasing the threshold increases precision and lowers recall, while decreasing the threshold does the reverse.
- We recommend using OCaMOSS with a threshold between 0.4 - 0.6, where 0.6 is less likely to detect plagiarised files, but is also less likely to detect false positives.

Self-generated test cases (descriptions are in README)
Description: 10 OCaml, 1 txt, 1 Java, 1 Python, 1 C (14 total test sets)
Total Instances of Plagiarism across all test sets: 13
Total Instances of Plagiarism caught by OCaMOSS (threshold = 0.4): 13

Score threshold	Precision	Recall	F1 Score
0.4	**100%**	**100%**	**100%**
0.6	100%	84.62%	91.67%
0.8	100%	61.54%	76.19%

Natural Language Processing Group, NUS SSID Demo Dataset (SSID_demo)[3]
Number of files: 10 (Java)
Total Instances of Plagiarism: 5
Total Instances of Plagiarism caught by OCaMOSS (threshold = 0.4): 5

Score threshold	Precision	Recall	F1 Score
0.4	**100%**	**100%**	**100%**

[3] https://github.com/WING-NUS/SSID/tree/master/doc/demo

0.6	**100%**	**100%**	**100%**
0.8	100%	60%	75%

Source Code Plagiarism Test Sets (MiniFactorialTests)[4]
Number of files: 20 (Java)

- Original: 1
- Type-1 (Exact copy/whitespace, layout, comments differences)[5] : 3
- Type-2 (Type, identifier, literals differences) : 2
- Type-3 (Reordering, adding/removing statements) : 10
- Type-4 (Same computation done by different syntactic variant) : 4

Total Instances of Plagiarism : 19
Total Instances of Plagiarism caught by OCaMOSS: 13

Detailed breakdown of results for threshold = 0.4:

Type of Plagiarism	**Precision**	**Recall**	**F1 Score**
Type 1	100%	100%	100%
Type 2	100%	100%	100%
Type 3	100%	70%	82%
Type 4	100%	25%	40%
Overall	**100%**	**68.4%**	**81.2%**

Note: Our system is not designed to catch Type 4 plagiarism, as we're only looking for blatant cases of plagiarism, and it is unclear that a different syntactic variant of a file is that. Furthermore, our effectiveness is limited by the fact that we only catch exact matches of code (Outside of variable name changes and whitespace removal), since we realise that when it comes to plagiarism detection and the consequences that arise in the real world because of it, it is a lot more harmful to have false positives than it is beneficial to slightly increase the chances of catching another file.

SOCO Training Corpus (SOCO_java/SOCO_c)[6]
Description: set of 258* Java files (1.5mb total) and 79 C files (352kb total) - we used these to test performance. They were not included in the submission due to excessive size.

Runtime for 258 Java files	36.0 seconds
Runtime for 79 C files	2.65 seconds

[4] https://github.com/nordicway/SourceCode-Plagiarism-TestSets
[5] Roy et al. https://www.cs.usask.ca/~croy/papers/2009/RCK_SCP_Clones.pdf (pg 4)
[6] http://users.dsic.upv.es/grupos/nle/soco/corpus.html

*The original SOCO set had 259 Java files, we removed file 063. See Known Bugs section for details.

Known Issues

1) This system suffered a stack overflow error when running file 063 in SOCO_java. Thus, we can only guarantee that our system will work for files that are individually 26kb and smaller, and may fail for files 63kb and larger. Sets of files totaling more than 1.5mb have not been tested, so there is no guarantee that our system will work for larger sets of files.

Division of Labor

Robert - Worked on most of preprocessing, including noise removal (the procedure of which includes white-space removal, comment removal, variable name replacement, and retrieval of relevant language information from JSON files), and k-gram generation, and all unit tests regarding preprocessing.

Gauri - Worked on the main insert operation in our two-three tree implementation and OUnit tests for dictionary module. Added test cases for groups of files, texts, and online data sets.

Danny - Implemented winnowing module, worked on REPL (basic functionality, pretty-printing, integrating the other modules into the interface) and parts of preprocessing (rehashing to display matching patterns during results generation, modifications to initial lexing to fix bugs). Added support for Python, Java, C.

Aniroodh - Worked on all of comparison and OUnit testing for comparison. Wrote some of the earlier functions in dictionary. Worked on the REPL, in particular with parsing through a directory, and printing out results after running a query. Worked on the end-to-end tests, and wrote the evaluation portion of the report.

Together - Worked on self-generated tests and most of the report.