Plagiarism Checker

Navneel Singhal

August 2020

Description

The assignment was to implement a plagiarism checker in C.

Usage

To compile the code, run the command `make`.
To run the code, run `./plagChecker <LOCATION_OF_TEST_FILE> <LOCATION_OF_CORPUS_FOLDER>`.
Important Note: When passing the location of the corpus folder, ensure that it ends with a /
This code has been tested to run on Linux (Ubuntu 20.04), and couldn't be tested on Windows due to the lack of a Windows machine. However this will work on POSIX-compliant machines.

The Model

Introduction to N-grams

A concept widely used in natural language processing, word N-grams are essentially "phrases" of length N in a given text, for our purposes. Their main importance is in the fact that they can represent Markov chains of length N, which allows for a simple representation of phrases which is also natural fit for languages like English, because the main structures that stand out in a text are linear owing to the low nesting-levels in most languages. Their relevance to our use-case is that in a plagiarized document, the frequency of any particular phrase should be heuristically similar to that in the document where it was plagiarized from.

High-level Description of the Algorithm

From any given text, we find the sorted array containing all possible 3-gram hashes in the text. To find the similarity between the texts, we extract frequency vectors of 3-grams corresponding to both texts, sorted according to 3-gram hashes in both the texts (and if a hash is not found in one file, we simply assign the value 0 to the frequency of that hash), and find the cosine similarity between any two such vectors. This is going to be the measure of similarity we will use in this implementation.
Cosine similarity for two vectors $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ is defined as

$$\mathcal{S}((a_1, \ldots, a_n), (b_1, \ldots, b_n)) = \frac{a_1 b_1 + \cdots + a_n b_n}{\sqrt{(a_1^2 + \cdots + a_n^2)(b_1^2 + \cdots + b_n^2)}}$$

The reasoning behind this is as follows.

1. For relatively more frequent 3-grams, the weight of that feature should be intuitively high, and sparse features are quite unimportant.

2. This measure is symmetric between the corpus and the queried file, so it doesn't introduce a bias between who plagiarised whom.

3. It is resistant to scaling; i.e., if there is a small document which has been plagiarised from a larger document, the results are similar to those if the smaller document was slightly larger.

To make feature extraction robust and resistant to the usage of different cases, we tokenize the text with respect to whitespace, commas, full-stops and newline characters, and then for each of the extracted strings, we convert them to lower case. To make feature extraction simpler, we use a polynomial hash (with base 127) for each string, and for each 3-gram (which is now a triple of integers modulo 10^9+7), we use a different polynomial hash (with base 997 so as to not bring in any unwanted correlation between concatenated words or something else of this sort).
We also remove all words whose length is less than 3, so as to remove commonly used words from consideration.

Working of Code

1. `long long int stringHash(char* s)`: This function computes the hash of a string as mentioned above. Time complexity is $\mathcal{O}$(length of string), and space complexity is $\mathcal{O}(1)$.

2. `long long int ngramHash(long long int a, long long int b, long long int c)`: This function computes the hash of a 3-gram, as mentioned above. Time complexity is $\mathcal{O}(1)$ and space complexity is $\mathcal{O}(1)$.

3. `void sort(long long int* a, int l, int r)`: This function sorts the subarray `a[l..r]` using merge-sort. Time complexity is $\mathcal{O}(r-l+1)$ and space complexity is $\mathcal{O}(1)$ (since we use a global buffer).

4. `long long int* workFile(char* fileName)`: This function returns a pointer to the sorted array containing the hashes of all the 3-grams in the file whose name is `fileName`. Time complexity is $\mathcal{O}$(total length $+ N \log N$) and space complexity is $\mathcal{O}(N)$ where N is the number of words in the file (we reuse the buffer used in the previous function).

5. `float findSimilarity(long long int* a, int n, long long int* b, int m)`: This function computes the frequency arrays for the arrays pointed to by `a` and `b`, and then finds the cosine similarity between them and returns the similarity as a ratio between 0 and 1. Time complexity is $\mathcal{O}(n+m)$ and space complexity is also $\mathcal{O}(n+m)$.

6. `int main(int argc, char* argv[])`: This is the main function, and here we traverse over all files in the given corpus directory, and find the similarity between the queried file and the corpus files, one corpus file at a time. For the time complexity, suppose the number of corpus files is t, the total number of characters in all corpus text files and the queried test file is s, the total number of words in file i is n_i, with the queried file being file 0, and the rest being corpus files. Then the time complexity is $\mathcal{O}(s + \sum_{i=0}^{t} n_i \log n_i + tn_0)$. The space complexity is $\mathcal{O}(n_0 + \max_{i=1}^{t} n_i)$, apart from a global buffer.