PhUSE US Connect 2018

CT13

Upholding Ethics and Integrity: A macro-based approach to detect plagiarism in programming

Praveen Kumar, Ephicacy, Bangalore, India
Sridhar Vijendra, Ephicacy, Bangalore, India

ABSTRACT

Good Clinical Practice is the fundamental guiding light for our industry. Ethical conduct ensuring patient safety and data integrity is crucial and as statistical programmers, it is our onus to conform with regulatory guidelines, processes and standards. Double programming is a mandated, routine activity to ensure an error-free delivery. Although a rare occurrence, there are instances of production or validation code plagiarized from one another that can have serious consequences. Identifying these instances well in advance and ensuring they were not intentional is critical for ensuring regulatory compliance and preventing loss of overall quality. A manual check ensuring that production and validation codes are not photocopies of each other is a tedious task for large number of programs and in contrast, automated detection can provide better quality with less bias. In this paper, we describe a simple validated macro that can identify any potential instance of plagiarism during double programming.

INTRODUCTION

Statistical programmers are constantly being reminded of the importance of their role in patient safety. Good programming practice and the various regulatory guidelines all have the bottom line of ensuring patient safety during study conduct. Statistical programming is connected quite explicitly to patient safety when it comes to validation of double programmed outputs. Independent double programming is done for a reason and that is to ensure accurate reporting of clinical trial results. However, on occasions when an erroneous data point or report slips through the programming team, they are caught by the statistician, medical writer or further down the line before it reaches the regulatory authorities. But what happens when the QC programmer does not do their job as a "validator" and instead simply re-creates the output using an exact copy of the production programmer's code? There is a good chance the report generated by the production programmer is indeed accurate and nobody ever knows of the misconduct. But what if new data results in an inaccurate report that lands up with the clinician or medical writer and the results are interpreted to mean something other than what is?

There is plagiarism in the statistical programming world. There have been instances, however rare, of code being copied by the QC programmer to **match** the results produced by the production programmer. This is not only unprofessional but also shows a lack of integrity that is also sadly reflected upon the organization employing the said programmer. Disciplinary action is always taken on the accused for the serious transgression but the short and long-term consequence of such an incident for the organization or the sponsor is far greater than what the programmer can imagine. If the programmer works for a CRO, there is a good chance the CRO will end up losing the contract with the sponsor.

Some organizations adopt simple strategies to prevent plagiarism. One of them is to have separate folders for main and QC programming, in addition to having separate teams for each to ensure there is no chance the main side programmers can even look at the QC side code and vice-versa. But the reality is - given tight timelines and perpetual resourcing constraints, not every sponsor or CRO can afford to adopt this strategy for every study that is programmed. The other strategy to combat plagiarism is to perform random checks on pairs of main and QC codes. This activity can be taken up by the lead programmer or a senior resource in the team. There are many ways to check codes in pairs – using line-by-line comparisons in an editor or using tools such as WinMerge® or ***diff*** in UNIX®. Such tools are not exclusively meant for this purpose, but they do the job. The only serious constraint of this approach is the lack of billability of this tedious activity. This activity is best done by senior programmers in the team who are always better occupied doing something more productive in terms of billability.

In this paper, it is suggested that the task of identifying plagiarism (code copying) between pairs of codes can be accomplished using a simple utility SAS® macro. This can reduce the burden of the programmer in the team given the task of reviewing and/or comparing codes and thereby improve overall quality within the organization without leaving anything to chance. This paper describes the approach taken to tackle plagiarism, explains the various aspects considered to build the macro, followed by some of the preliminary testing done on the macro and finally, summarizes the available results and proposes future work.

AVAILABLE TOOLS AND THE GOLD STANDARD

The comparison of a pair of codes to find any hint of plagiarism cannot be done by a tool or an algorithm that finds differences but instead must be done by an algorithm that finds similarities. Obviously, what is not similar is different

but quantifying similarities is the cornerstone of the problem we are trying to solve with this utility. Checking plagiarism in assignments submitted by students of university level programming classes is done routinely to ensure originality. Tools such as MOSS®[1] are available as a service to automate this task but this tool does not support SAS code. The software engineering industry is also not new to the concept of copied code, although the main concern of the software industry is more about that of duplicated code that causes redundancy and complicates code maintenance. There is no widely available tool specifically used for checking of code copied between the main and QC side but there is no doubt that the current gold standard to identify copied code is through manual checking. Regardless of the complexity of the algorithm to find the similarities, it is hard to incorporate a level of fuzziness and intuition that human intervention can bring to this activity. Hence, in the rest of the paper, for all comparisons to the results of the proposed macro, the manual code copying result will be used as the gold standard reference.

BUILDING THE MACRO

The high-level flowchart of the proposed macro is shown in **Figure 1.** Once the main and QC programs are read in as text strings into a dataset where each observation is a line of SAS code from the program, some pre-processing is done on the strings. The pre-processing includes removal of the program header and discarding of blank lines from both codes. The next step would be derivation of a couple of features each from the programs. The features are described in detail in the next section. Each of these features will produce a quantifiable result. A threshold will be set for each of these features and they will be used to determine if the codes are copied from each other or not. To determine the thresholds, the features were run on a preliminary set of "training" inputs after which the macro was tested on a set of "test" inputs. The decision rule applied for each feature is shown illustrated in **Figure 2**.

Figure 1: High-level flowchart

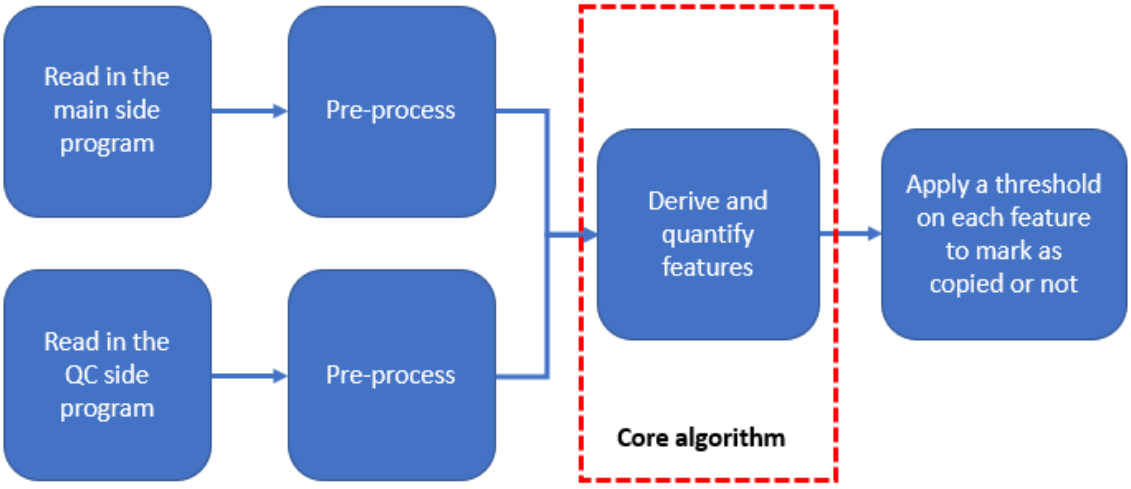
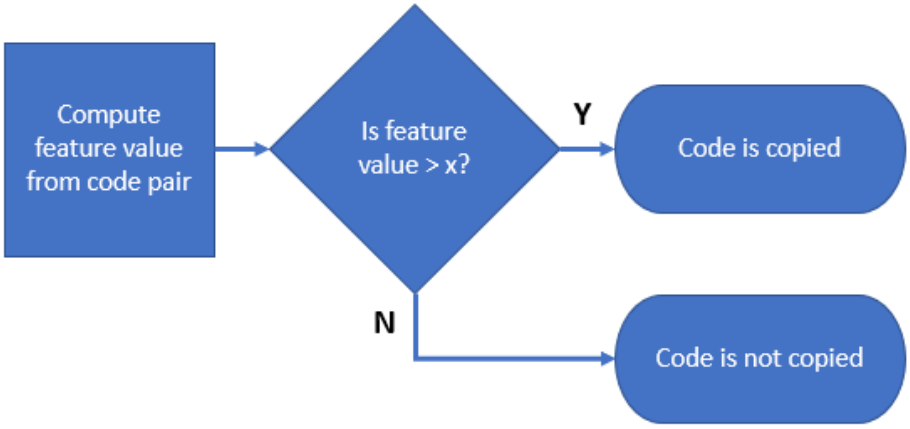


Figure 2: Decisions made using each feature



THE CORE ALGORITHM

One of the best ways to identify an instance of copied code is to think like a programmer who wishes to copy code and identify situations that would call for an entire program to be copied. Why would a programmer copy a program

instead of writing it fresh? Would it be to save time, or would it be because they are not qualified to write the program of that complexity? If a programmer who wishes to copy the code from the production to the QC side or vice-versa does so to save time, it probably means they are smart enough to know that they should make the copy look quite unlike the original they have copied from. This means that they are probably going to change intermediate variable names, intermediate data set names and probably some of the comments in the header of the program. They are also probably going to keep the intricate variable derivations as is because after all, that is the most time-consuming part of writing fresh code. If they are smart, they will also switch some of the DATA steps for PROC SQL or vice-versa leaving the flow of the program as it originally was. The other possibility of a programmer copying code is because they don't know how to write the code they are supposed to. In this case, the programmer would most likely create an exact replica of the code and leave it almost like the original.

The core algorithm involves extraction of specific features from the post-processed text strings that comprise the main and QC codes. Based on the above discussion, several features were considered:

- Number of similar lines of code
- Intermediate data set names
- Number of intermediate data sets
- Intermediate data set variable names
- User-defined macro names
- Number of PROC SQL steps vs. number of DATA steps
- Number of characters in comments that are similar

Of those listed above, the following 3 features were picked for testing:

FEATURE 1: SIMILAR LINES OF CODE

Feature F1 depicted in **Equation** 1 simply computes a percentage of the number of lines of code that are common between the two programs. The denominator for the percentage is the maximum number of lines of code in both codes. When a code is compared against itself for this feature, a value of 100% is produced. This feature is derived in SAS using a simple merge between the main and QC side string data sets where each observation is a line of SAS code from the respective programs. This ensures a simple line-by-line comparison that mimics the line-by-line comparison done during a manual check.

Equation 1: Formula for Feature F1

$$F1 = \frac{\text{Number of lines of exactly matching codes}}{\text{Maximum number of lines in both codes}} \times 100$$

FEATURE 2: NUMBER OF DATA STEPS AND PROC SQL STEPS

Feature F2 obtains a ratio of the number of DATA and PROC steps in the main and QC side programs. A ratio of 1 or a percentage of 100% in this feature indicates that there are an exact number of PROC and DATA steps (combined) in both programs indicating a chance of the code being copied. A depiction of this feature can be seen in **Equation 2**.

Equation 2: Formula for Feature F2

$$F2 = \frac{\text{Number of PROC and DATA steps in main side}}{\text{Number of PROC and DATA steps in QC side}} \times 100$$

FEATURE 3: COMMON DATASET NAMES

Feature F3 computes a percentage of the number of common intermediate data set names out of a maximum number of data sets in both programs as shown in **Equation 3**.

Equation 3: Formula for Feature F3

$$F3 = \frac{\text{Number of common intermediate data set names}}{\text{Maximum number of intermediate data sets in both codes}} \times 100$$

RESULTS OF THE ANALYSIS

For examining the feature values generated by the above-mentioned algorithm, we used main and QC code pairs from actual studies. As mentioned earlier, a manual check is considered the gold standard reference for this exercise and the results of this macro will be compared with the results of a manual check. We had a senior programmer in the team examine every pair of codes. We didn't have any available plagiarized code pairs and to generate a suitable set of plagiarized inputs for testing, we created copied code pairs that would replicate real plagiarized inputs. We managed a total of 36 pairs of codes for checking with the macro, of which 12 pairs were plagiarized and the remaining were not.

To ensure effective testing of the features, we divided the 36 pairs into a training set of 24 pairs and a test set of 12 pairs, with each set containing enough number of copied and uncopied pairs. The objective was to use the training set to identify the threshold percentages for each feature and then test the corresponding thresholds with the test set.

Figure 3, **Figure 4** and **Figure 5** show the results of features F1, F2 and F3 in the form of scatter plots. These results have been obtained for the 24 pairs of training inputs. The red dots indicate the values of the copied codes and the blue dots indicate the values of the code pairs without plagiarism. We attempted to arrive at a threshold value for each feature by adjusting the threshold such that the copied and uncopied pairs can be separated successfully by that threshold.

Figure 3: Scatter plot of the results of Feature F1

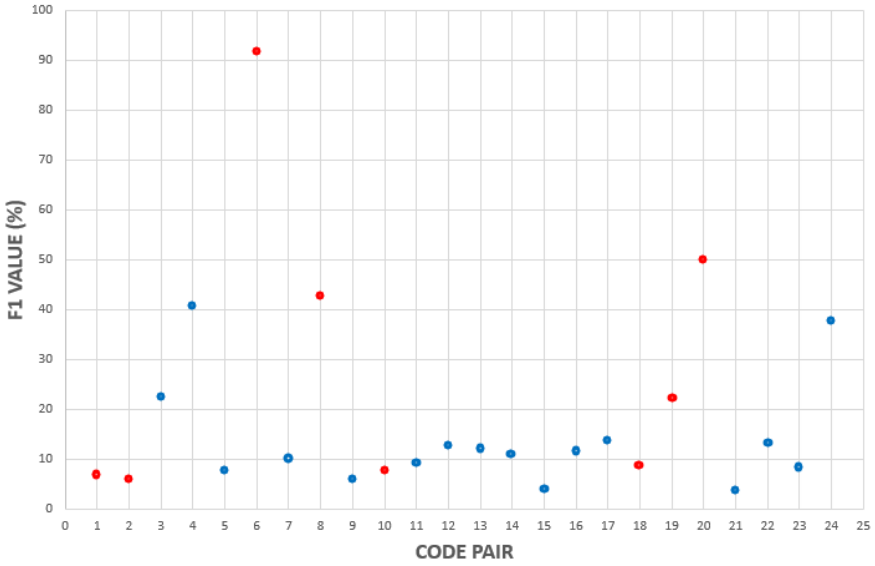


Figure 4: Scatter plot of the results of Feature F2

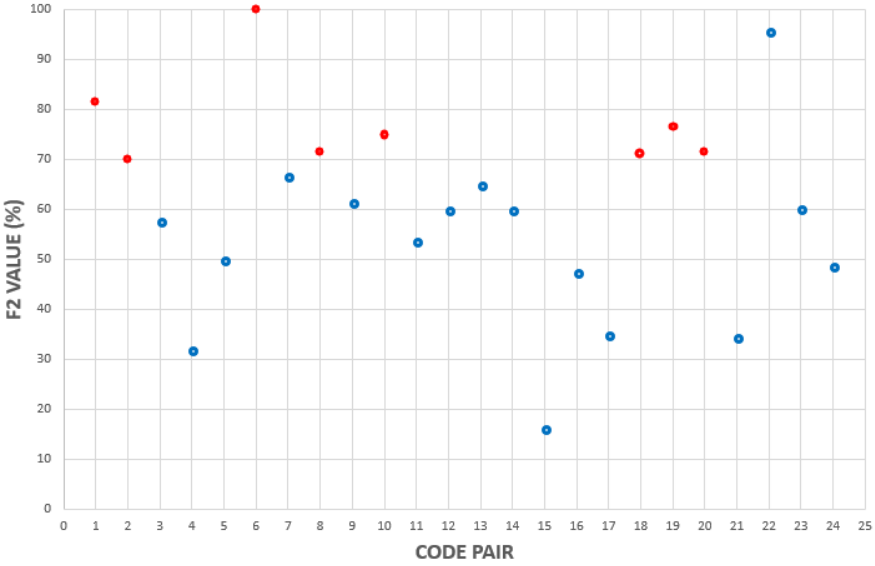
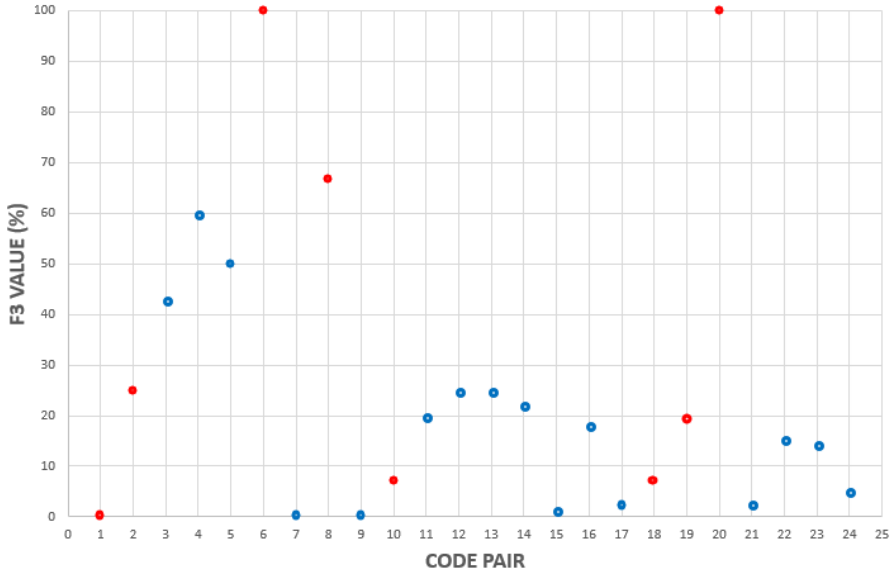


Figure 5: Scatter plot of the results of Feature F3



From the scatter plot of feature F2 (**Figure 4**), it is possible to identify a fairly clear demarcation between the red and blue dots. Most of the uncopied codes (blue dots) except for one (code pair 22) lie below a threshold of around 70%. However, it does not look feasible to distinguish a very clear threshold for features F1 and F3. If we try to set a threshold of 20% for feature F1 in **Figure 3** and suggest that all copied codes will have a feature F1 value of >20%, that does not appear to demarcate all the 24 training inputs correctly. A similar scenario presents itself for feature F3 and a threshold of 20% (>20% of dataset names common between copied codes) seems to accurately select only 4 of the 8 copied codes in the training set. For testing the macro on the test set, a threshold value must be selected for each feature. From the training set, this threshold is selected as >20% for copied codes for feature F1, >70% for copied codes for feature F2 and >20% for copied codes for feature F3.

The macro was then run on the test set of 12 pairs and the threshold values derived using the training set for each of the features were applied on the 12 test inputs. The results of this test for feature F1 are shown in **Table 1** in the form of a simple sensitivity-specificity table. Feature F1 has correctly classified only 5 of the 12 input pairs in the test set. Similar results for features F2 and F3 are shown in **Table 2** and **Table 3** respectively.

Table 1: Test Results for Feature F1

	Code copied (as per Gold Standard)	Code Not Copied (as per Gold Standard)
Code Copied (as per macro)	1	4
Code Not Copied (as per macro)	3	4
Total	4	8

Table 2: Test Results for Feature F2

	Code copied (as per Gold Standard)	Code Not Copied (as per Gold Standard)
Code Copied (as per macro)	3	0
Code Not Copied (as per macro)	1	8
Total	4	8

Table 3: Test Results for Feature F3

	Code copied (as per Gold Standard)	Code Not Copied (as per Gold Standard)
Code Copied (as per macro)	1	4
Code Not Copied (as per macro)	3	4
Total	4	8

CONCLUSION

With stringent regulations and guidelines such as GCP paving the way for patient safety in drug trials, it is important for us as statistical programmers to abide by them and keep patient safety in mind always. One of the critical aspects of programming that can compromise patient safety and have possibly cumulative debilitating consequences is an incident of the QC program being an exact copy of the main program or vice-versa. Such incidents are rare but it is important to take preventive action to ensure high levels of quality and integrity.

Since manual checking of instances of copied code is a tedious and almost always a non-billable activity, it is suggested that automated checking of plagiarized code be done with the help of a SAS utility macro. A simple macro that derives specific features from the main and QC side codes was developed and this macro was tested using actual non-plagiarized study programs and artificial plagiarized inputs. Using the thresholds computed from the training set of inputs, the macro was tested for sensitivity and specificity on a new set of test inputs. Only 5 of the 12 set of inputs were identified correctly by feature F1 indicating a very poor sensitivity of 25% and a specificity of only 50%. However, feature F2 fared much better with a sensitivity of 75% and specificity of 100% for the test inputs. Feature F3 was not any better compared to feature F1 and identified only 5 of the 12 sets of inputs correctly. Taking a majority (2 or more) of the 3 features to arrive at a decision does not seem to improve the results any more than what feature F2 can do. It is interesting to note that feature F2, which looks at the ratio of the number of PROC and DATA steps in the main and QC side programs is such a promising indicator of the similarity between the codes. It is possible that feature F1 can provide better results if the merge between the main and QC side code strings is optimized to improve detection of similarity.

The results provided by the macro are promising for feature F2 but this simple macro is just the tip of the iceberg in terms of a fully operational code checking macro. The test set used here is way too small and far less varied to prove the efficacy of such a macro for a wide range of coding styles and program types. Along with more extensive testing of the macro, the macro needs to be enhanced further with additional features and possibly a more complex algorithm. Algorithms to detect and quantify similarities between strings and text files are a widely researched topic and it will be very interesting to see the results that more complex algorithms can produce. Since we are comparing programs and will most likely have access to the logs of the same programs, the logs can provide additional information about any plagiarized code to complement the information given by the programs.

In any case, tools such as these should be primarily used for screening of programs while it is advisable to manually look through any positives that the macro throws up to confirm that plagiarism has indeed happened. Even with a tool with a high sensitivity and specificity, a final manual check will be necessary in the rare possibility that 2 pieces of code on the main and the QC versions look very similar simply by pure chance. It is possible that the two programmers on either side were mentored similarly or learnt programming together or simply write programs in a similar style. All said and done, nothing can better a meticulous manual comparison of codes.

REFERENCES

1. Aiken, Alex. http://theory.stanford.edu/~aiken/moss/. [Online]

ACKNOWLEDGMENTS

Thanks to the Ephicacy management for their unwavering support in making this happen and a special thanks to the Ephicacy Biostatistics & Programming team members who provided their time and valuable feedback during the writing of this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Praveen Kumar
Ephicacy Lifescience Analytics
Bangalore, India
praveen.kumar@ephicacy.in

Sridhar Vijendra
Ephicacy Lifescience Analytics
Bangalore, India
sridhar.vijendra@ephicacy.in