Paper SS01

Fishing for Code Plagiarism in Clinical SAS Programming

Menaga Guruswamy Ponnupandy, ICON plc, PA, USA

ABSTRACT

Quality control is the heart of clinical SAS programming, and it is entrenched through double programming. As widely claimed, it is a standard method where two programmers write code from scratch for the same specification to compare their outputs. If the output matches, then it proves that their respective program logics are built unique from the specification, but this practice of double programming can result in code plagiarism as there is no hard and fast rules or restrictions set on the .sas files to prevent a programmer from looking into the other programmer's code. In such cases the so-called golden standard is unsuccessful in producing a quality output. This paper will walk through the measures that can intercept such instances. These measures can vary from a basic grep or awk command to an advanced script specifically designed to capture problematic code blocks of one program that closely resembles with another.

INTRODUCTION

When working with clinical SAS programming, it is important to be aware of the potential consequences of code copying. While copying code may seem like a quick and easy way to get the desired results, this practice can lead to quality or compliance issues down the line. For example, if a programmer copies code from another programmer who builds the same logic, then it can produce erroneous results. This type of error often goes undetected until it is too late and can cause irreparable damage.

Ultimately, it is best to avoid copying code altogether but if it happens how do you catch it being a study lead or code auditor? Most of the software development process relies on some sort of source control management (SCM) system to track changes to code over time. These systems allow developers to keep track of who changed what lines of code when, and why. This can be extremely helpful when trying to figure out who broke something, or when trying to understand the evolution of a particular piece of code. Each SCM system has its own way of storing this information, but the general idea is the same. When a developer makes a change to a file, they 'commit' that change to the SCM system. This creates a new 'revision' of the file that includes the changed lines, as well as some metadata about who made the change and why. The SCM system then tracks all future changes to the file in subsequent revisions.

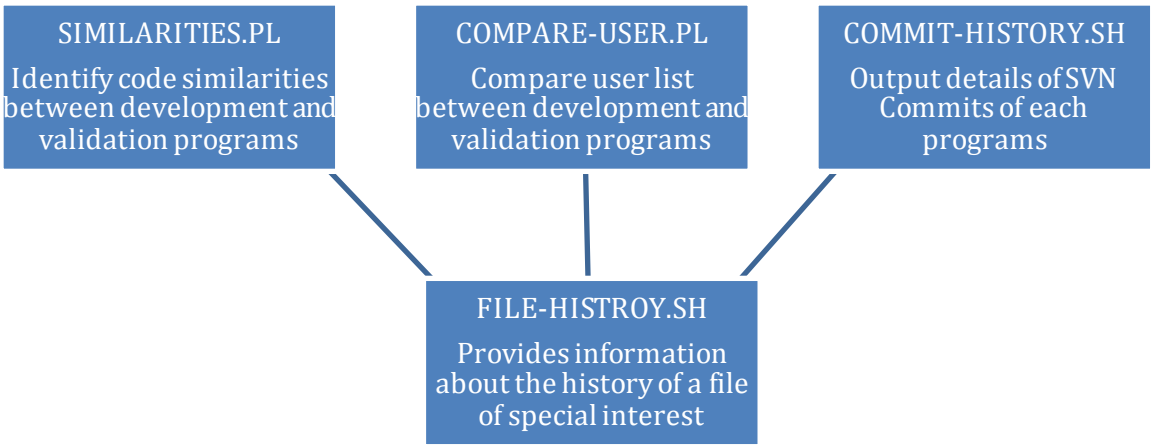
For this reason, it's important for code auditors to have a basic understanding of how SCM systems work and how to interpret the information that they provide. With this knowledge, reviewers can build or use automated tools more effectively, and can also perform manual audits when necessary to identify potentially plagiarized code.

Taking steps to prevent code plagiarism is essential to maintain high-quality standards in clinical SAS programming.

TOOLS TO PERFORM CODE AUDIT

Traditionally, code review has been a manual process but now there are a variety of approaches that can be used for code audit. The approach described in this paper makes use of Subversion (SVN) and Perl or Shell scripting techniques. SVN is a version control system, a subset of SCM that helps to manage and track changes to code and assets across projects. Perl or Shell scripting is a versatile that it can be used for automating tasks, writing small utilities, or prototyping new ideas. The scripts described in this paper can be easily customized to run on a variety of different platforms and or version control systems to meet project specific needs. By using SVN and Perl, programmers can create a powerful code review tool that is easy to use and maintain. There are many resources available online to get started.

In this paper, we will discuss about scripts that can help the end-user perform a code audit. The first script, similarities.pl, can be used to detect common lines between two files. This can be useful in identifying duplicated or identical code blocks between development and validation. The second script, compare-user.pl, can be used to identify programs where the same programmer has worked on both the development and validation codes. This is important because it can help in preventing issues caused by incorrect assignments and code check-ins. The third script, commit-history.sh, can be used to output the details of each SVN commits of a specific program. This can be helpful in retrieving information about number of programmers who worked on a specific code and date and time they've checked in the codes. In closing, the script file-histroy.sh can help end-user dig into the hidden details of a plagiarized code.



SIMILARITIES.PL

The Perl script similarities.pl will compare the development and validation codes and print the common lines between two files to an output file. If there is no similarity between the two files, or if the relevant file for comparison is not found in the QC directory, then no output file will be created. The common lines like "end; quit; run; blank lines" if present in both is eliminated from the third output file for better visualization of matching lines. This comparison can be recursively performed for all files within a directory, simply by running the script. Here are the steps to execute the script –

1. To run this script, the user will first copy the similarities.pl to the chosen directory ideally one level above where the files are located.

2. The user will then open Putty and navigate to the chosen directory. For example, cd /projects.

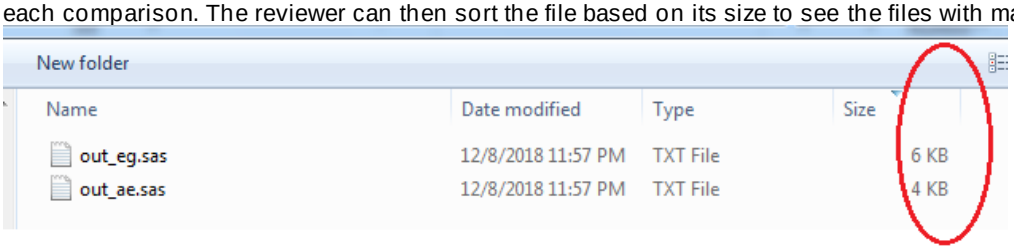
3. Once in the correct directory, the user needs to ensure that there are three physical paths that exists to run the script successfully. The script is designed to throw an error if the user doesn't provide all three arguments in the command line or if any one of the directories does not exist.

    - -main (to provide the path of the development program)
    - -qc (to provide the path of the validation program)
    - -output (the path to re-direct the final text results of comparison)

4. Run the script by using the command and by providing the paths needed for each argument:

    "perl similarities.pl -main /projects/prog -qc /projects/qcprog -output /projects/qcprog/matches"

5. This execution will generate an output file named "out_programname.sas.txt" under /projects/qcprog/matches for each comparison. The reviewer can then sort the file based on its size to see the files with maximum similarity.



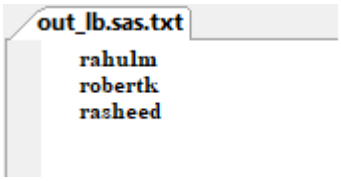
COMPARE-USER.PL

The compare-user.pl is similar to the script 'similarities.pl' however the application is unique. The compare-user.pl is used to compare the user list between each of the development and validation programs within a directory and to output programmer names that are common between both. It compares the SVN log of development and validation program and produces the output file named "out_programname.sas.txt" with programmer names who had checked in or worked on both the development and validation programs. In an ideal case, a programmer is expected to work on or check in codes only on one side and are not supposed to work or check in on the other side. This application is essential for ensuring that there is no cross-contamination between development and validation programs, which can lead to issues further down the line.

1. To run the script, follow the same instructions from step 1 to 3 of similarities.pl and pass on the command with three arguments. The end-user may want to edit the physical path of the 3rd argument for clarity i.e., from /projects/qcprog/matches to /projects/qcprog/common_user. Run command: -

   "perl compare-user.pl -main /projects/prog -qc /projects/qcprog -output /projects/qcprog/common_user"

2. Successful execution will create the output file "out_programname.sas.txt" under /projects/qcprog/common_user. In the below example, the common users between development and validation for the lb.sas are listed under /projects/qcprog/common_user/out_lb.sas.txt



COMMIT-HISTORY.SH

The commhist.sh is a simple shell script used to output details of SVN commits of all programs under development and or validation folder.

```
#!/bin/sh
for f in *.sas
do
echo "processing $f ... \n"
svn log $f --quiet | grep "^r"

done
```

For new line

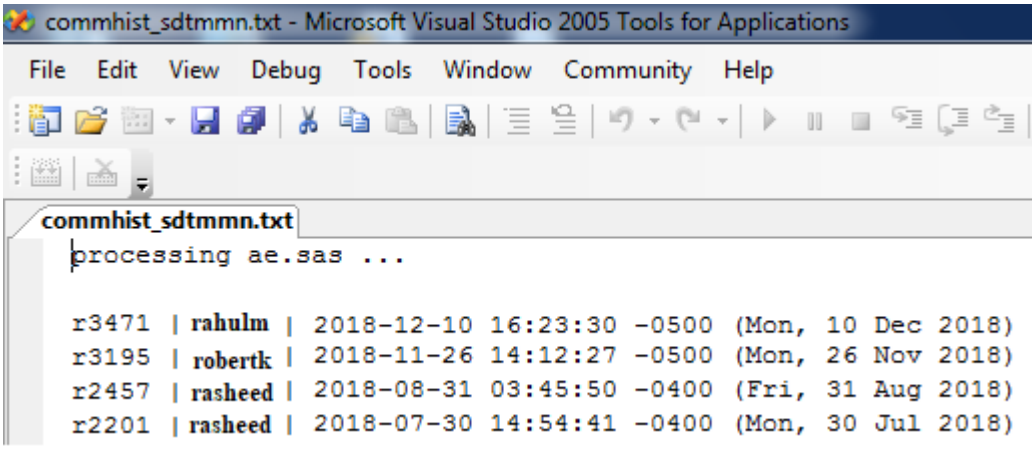
For readability

1. To execute the script 'commit-history.sh', save the above as commit-history.sh to '/projects/prog' the location of the development programs. Now, run command –

   "commit-history.sh > commhist_sdtmmn.txt".

2. As the user runs the script, the commit history report is generated with an output file named 'commhist_sdtmmn.txt' within the same folder.

3. The output file commhist_sdtmmn.txt contains the revision number, user id and the date the update has been made. These commit history details are presented not just for one code but for all the code under a specific directory '/transfer/prog'.



FILE-HISTORY.SH

By running the similarities.pl script on a terminal, a reviewer can quickly generate a report showing the similarities between codes in two folders and thus warrant closer scrutiny. The size of the output file from 'similarities.pl' can be used to judge the extent of similarities between two files. For example, if one file is significantly larger than the other, it is likely that there are more similarities between them. This information along with the information generated from the two other scripts can be used to prioritize review efforts and ensure that the most important files are reviewed first. If the code reviewer has itemized the most important file or the file of special interest, then they can run the file-history.sh script provided in the appendix of this paper or by using the commands below –

- svn log --diff vs.sas > vs_chghist.txt
- svn log --diff qc_vs.sas > qc_vs_chghist.txt

These .txt files generated from the above commands can provide a complete history of a file of special interest. By looking at the history of a particular file, it's possible to see exactly who changed what, and when. This can be very helpful in tracking down who introduced a particular bug, or in understanding why a particular piece of code was added. Additionally, auditors can use this information to identify potentially plagiarized code, or code that has been copied from development to validation or vice versa. This process is extremely important in auditing code for errors or security issues.

CONCLUSION

A code audit is an important part of quality assurance for any clinical programming project with quality issues. By carefully examining the code, a reviewer can identify potential problems. However, manually reviewing a large codebase can be a daunting task. Fortunately, there are tools that can help. To conclude, tracking is doable, as we can always go back to who made what changes or who had checked in a particular line of a file in SVN. Therefore, code audit must be a part of Best Practices as and when needed to Avoid Code Plagiarism.

ACKNOWLEDGMENTS

I would like to express deepest appreciation to my family, especially my sister, brother, and spouse.

CONTACT INFORMATION

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

Author Name: Menaga Guruswamy Ponnupandy
Company: ICON US
Address: 731 Arbor Way Ste 100 Blue Bell
City / Postcode: PA/ 19422-1987
Email: Menaga.GuruswamyPonnupandy@iconplc.com
Web: www.ICONplc.com

Appendix: Scripts

similarities.pl

```
#!/usr/bin/perl

use strict;
use Getopt::Long qw(GetOptions);

sub main {
  my ($dir1, $dir2, $outDir);
  GetOptions('main=s' => \$dir1, 'qc=s', \$dir2, 'output=s', \$outDir)
      or usage();

  # Check if the user has provided all the required inputs
  if (! $dir1 || ! $dir2 || ! $outDir) {
    print "ERROR: Please provide all arguments\n";
    usage ();
  }

  # Check if all the directories exist
  if (! -d $dir1 || ! -d $dir2 || ! -d $outDir) {
    print "ERROR: Please make sure that all the required directories exist \n";
    exit (1);
  }

  # Read all files from the main dir
  opendir my $DIR, "$dir1" or die "Cannot open directory: $!";
  my @files = readdir $DIR;
  closedir $DIR;

  # Process each file from the main dir
  foreach my $file (@files) {
    # Process only the .sas files. Also, Skip the special "." and ".." files.
    next if ($file =~ /^\.$/);
    next if ($file =~ /^\..$/);
    next if ($file !~ /\.sas$/);

    my $file2 = "$dir2/qc_$file";
    my $outFile = "$outDir/out_$file.txt";

    if (-e "$file2") {
      my @outputLines;
      print "Processing $dir1/$file and $file2 ... \n";

      # Read the main and the qc files
      open FILE1, "$dir1/$file";
      open FILE2, $file2;
      my @file2_lines = <FILE2>;
      foreach my $line (<FILE1>) {
        # Process each line of the main file
```

```
        # Tokenzie each line so that we can form a search string
        my @tokens = split (/\s+/, $line);
        my $searchString = "";
        foreach my $token (@tokens) {
          $searchString .= quotemeta ($token);
          $searchString .= '\s*';
        }

        # Search the qc file for this line using the formatted search string
        my @hit = grep (/^$searchString$/i, @file2_lines);
        if (@hit) {
          push (@outputLines, $hit[0]);
        }
      }

      # If @outputLines has any data, then we have matches
      if (@outputLines) {
        print "\t ... Matches\n";
        print "\t ... Output in $outFile\n\n";
        open (OUT, ">$outFile");
        #my @out = `grep -iFxf $dir1/$file $file2`;
        foreach my $line (@outputLines) {
          # Skip if the match is any of the following
          next if ($line =~ /^\s*end;\s*$/);
          next if ($line =~ /^\s*quit;\s*$/);
          next if ($line =~ /^\s*run;\s*$/);
          next if ($line =~ /^\s*$/);
          print OUT $line;
        }
      } else {
        print "\t ... NO matches\n\n";
      }
    } else {
      print "WARN: $file2 doesn't exist\n\n";
    }

  }

}

sub usage {
  print "Usage: $0 -main <main dir> -qc <qc dir> -output <Output dir>\n";
  exit;
}

main ();
```

compare-user.pl

```
#!/usr/bin/perl

use strict;
```

```
use Getopt::Long qw(GetOptions);

sub main {
  my ($dir1, $dir2, $outDir);
  GetOptions('main=s' => \$dir1, 'qc=s', \$dir2, 'output=s', \$outDir)
      or usage();

  # Check if the user has provided all the required inputs
  if (! $dir1 || ! $dir2 || ! $outDir) {
    print "ERROR: Please provide all arguments\n";
    usage ();
  }

  # Check if all the directories actually exist
  if (! -d $dir1 || ! -d $dir2 || ! -d $outDir) {
    print "ERROR: Please make sure that all the required directories exist \n";
    exit (1);
  }

  # Read all files from the main dir
  opendir my $DIR, "$dir1" or die "Cannot open directory: $!";
  my @files = readdir $DIR;
  closedir $DIR;

  # Read sas files
  foreach my $file (@files) {
    # Process only the .sas files. Also, Skip the special "." and ".." files.
    next if ($file =~ /^\.$/);
    next if ($file =~ /^\..$/);
    next if ($file !~ /\.sas$/);

    my $file2 = "$dir2/qc_$file";
    my $outFile = "$outDir/out_$file.txt";

    if (-e "$file2") {
      print "Comparing the user list of $dir1/$file and $file2 ... \n";
      # Run the svn command for file1 here
      my @f1_out = `svn log $dir1/$file --quiet | grep "^r" | awk '{print \$3}' | sort | uniq`;
      # Run the svn command for file2 here
      my @f2_out = `svn log $file2 --quiet | grep "^r" | awk '{print \$3}' | sort | uniq`;

      # Find intersection of two arrays
      my %isect;
      my %f1usr;
      foreach my $user (@f1_out) { $f1usr {$user} = 1; }
      foreach my $user (@f2_out) {
        if ($f1usr {$user}) { $isect {$user} = 1; }
      }

      my @common_users = keys %isect;
      if (@common_users) {
        print "Common users: @common_users \n";
        open (OUT, "> $outFile");
        print OUT "@common_users";
      } else {
```

```
        print "No common users\n";
      }
    }
  }
}

sub usage {
  print "Usage: $0 -main <main dir> -qc <qc dir> -output <Output dir>\n";
  exit;
}

main ();
```

commit-history.sh

```
#!/bin/sh
for f in *.sas
do
echo "processing $f ... \n"
svn log $f --quiet | grep "^r"

done
```

file-history.sh

```
#!/bin/bash

# Example:
#   file-history.sh vs.sas
#
# Outputs the full history of a given file as a sequence of
# logentry/diff pairs.  The first revision of the file is emitted as
# full text since there's not previous version to compare it to.

function history_of_file() {
    url=$1 # current url of file
    svn log -q $url | grep -E -e "^r[[:digit:]]+" -o | cut -c2- | sort -n | {

#       first revision as full text
        echo
        read r
        svn log -r$r $url@HEAD
        svn cat -r$r $url@HEAD
        echo

#       remaining revisions as differences to previous revision
        while read r
        do
            echo
```

```
            svn log -r$r $url@HEAD
            svn diff -c$r $url@HEAD
            echo
        done
    }
}
history_of_file
```