Be wise, plagiarize

Karma Tarap, Novartis Basel

Introduction

Standards are an essential component of Clinical Trial reporting. Pharmaceutical companies often invest considerable time and resources to develop standard programs and macros to help meet strict deadlines, whilst simultaneously ensuring high quality deliverables and keeping the resources required to a minimum.

In an idealized world, that would be the end to it – each output would have a standard program and we programmers would simply have to push the proverbial button. However, we have the fortune of working in a more complex world; a world where deviations from the standard – however minor – are common place. The result of all this is often 'standard' programs require considerable pre and/or post –processing. And in the more extreme cases where we don't have a standard program that performs the job sufficiently, the entire program will have to be generated from scratch.

So how can we quantify exactly how standard our code is? Such metrics would allow us to identify which standard programs need updating, replacing, or simply doesn't exist.

The goal of this paper is therefore to examine how we can use different techniques to quantify how often standard and non-standard programs are used, and to what extent. The proposed algorithms have been implemented in Proc Groovy (SAS v9.3).

Software Plagiarism Detection

In essence, to solve the 'standard similarity' problem, we want to examine and quantify the similarity between some program and the corresponding standard program. This problem has been extensively researched for detecting software plagiarism in both academia and industry. The main difference here is we are interested to find out where plagiarism isn't occurring.

A successful algorithm should be able to detect similarity on different levels, and also be efficient enough to allow comparisons of multiple files. Additionally, it must be able to withstand modifications made by the programmer that don't affect the final outcome of the program (such as changing comments, renaming variable or dataset names). Finally, it should be flexible to changes in ordering of statements and even procedures and data steps.

Why Groovy

Groovy is a dynamic language for the Java Virtual Machine with modern programming language features [3] that has been integrated into Base SAS (v9.3).

The decision to use Groovy was primarily due to the opinion that, as a general purpose language, it would be the best tool for the task. Additionally, it allowed us to experiment with this welcome extension of the SAS environment.

The naïve approach

In order to appreciate why we are using a more complex algorithmic approach, we need to first be aware of the consequences of using a simpler brute force approach.

A simple approach would be to scan over a file, and try to find whether each word exists in the corresponding standard file, and if so, update some counter.
This approach presents a few issues. First, due to the quadratic nature of the number or comparisons required, the algorithm would scale poorly: adding an extra word to file A would increase the number of comparisons required by the number of words in file B.

Another, equally important issue with this approach is the difficulty of getting a meaningful result. As we are matching words directly, this will not take into consideration modifications that don't affect the programs output. A further compounding issue is that by judging each word in isolation, we are losing the context of these words (Table 1).

The issues identified in this approach can be classified as follows:

1. Purpose – *The purpose of the word*
2. Context – *The context of the word given the surrounding words*
3. Ordering – *Changes of order of sections in a file*
4. Performance – *How long does it take to run, and how does it scale?*

The remainder of this paper will examine how we can improve on the issues identified in the naïve approach.

`proc sort data=class;`
`by age;`
`run;`
	`data class.proc ;`
	`sort = ' by age ' ;`
	`run ;`
		`/*proc sort`
		`data=class;`
		`by age run;*/`

Table 1: Under the naïve approach, all these programs would be considered a high scoring match

Tokenization and pre-tokenization:

So let's try to address the first issue. In programming languages, different types of words (tokens) have different meaning to the compiler, so this would be a natural way to classify the programs. We can also get rid of text we don't want to cover in our match, such as comments and spaces.

C-style multiline comments can be removed using a regular expression. As SAS allows for a multitude of line comments that can span multiple lines, these can get confused with algebraic expressions containing the * operator such as: x * y = z;. To handle this, we first break it down into SAS statements delimited by a semi-colon before further processing.

SAS has four basic token types [5]:

- **Literal** - One or more characters enclosed in single or double quotation marks.
- **Name** - One or more characters beginning with a letter or an underscore.
- **Number** - A numeric value.
- **Special character** - Any character that is not a letter, number, or underscore.

In addition we want to include the macro statements as a basic token type. We can now identify the basic token types with this regular expression:

```
re ='''(?si)(?:"[^"]*"|'[^']*')|%?[a-z_0-9]+|\\p{Punct}|\\d+|\\s+|.*'''
```

The SimpleTokenizer class presented in the appendix is then used to further subset these tokens into SAS macro and Base keywords. To increase the specificity of the tokenization step, this can be extended further.

Taking the following sample SAS code, we can now translate it into its lexical parts:

```
%let pgmname = aeder;
data aev0 ;
   set data_a.a_aev ;
   *For unscheduled visits, use the repeat visit number ;
   vis_1n = ifN(vis1n eq 999, rpevis1n, vis1n) ;
   *Compress out hidden characters from AE names;
   aevnam1a = compress(aevnam1a,,'kw') ;
   run;
```

Table 2: Example SAS program before tokenization

After tokenization, this becomes:

```
[mKeyword, bIdentifier, Operator, bIdentifier, bKeyword, bIdentifier,
bKeyword, bIdentifier, Punctuation, bIdentifier, bIdentifier,
Operator, bIdentifier, Separator, bIdentifier, bIdentifier, Number,
Separator, bIdentifier, Separator, bIdentifier, Separator,
bIdentifier, Operator, bIdentifier, Separator, bIdentifier, Separator,
Separator, Literal, Separator, bKeyword]
```

Table 3: Example SAS program after tokenization

As we only have 9 token types in our example, we will abbreviate this and store it as a list of integers:

Using the mapping:

1- Literal	4- bIdentifier	7- Operator
2- Number	5- mKeyword	8- Separator
3- bKeyword	6- mIdentifier	9- Punctuation

We get:

[5, 4, 7, 4, 3, 4, 3, 4, 9, 4, 4, 7, 4, 8, 4, 4, 2, 8, 4, 8, 4, 8, 4,7, 4, 8, 4, 8, 8, 1, 8, 3]

We now have our program text in a form that specifies the purpose of the words. As such, matches will be flexible to changes in variable and dataset names.

In order to address the second issue of context, we can group tokens with their surrounding tokens. This is done by first creating an n-gram representation of groups of tokens.

n-grams and Jaccard's similarity coefficient

An n-gram is simply a representation of a group of n tokens that can be thought of as the "fingerprints" of a section of code. So, if we use example n of 4, the token list:

5, 4, 7, 4, 3, 4, 3, 4, 9, 4, 4, 7

Can be chunked into the following sets of length 4 (4-grams).

{**5, 4, 7, 4**} {**3, 4, 3, 4**} {**9, 4, 4, 7**}

When we are looking for similarity, we will want to check the number of distinct n-grams in common between the two files. This can be represented by Jaccard's Coefficient, which is calculated by dividing the size of intersection between sets A & B by the size of their union:

(A, B) =$A \cap B / A \cup B$

Take for instance the following token sequence from two different code files:

File A:
{5, 4, 7, 4} {3, 4, 3, 4} {9, 4, 4, 7}

File B:
{5, 4, 7, 4} {3, 4, 3, 4} {3, 4, 5, 7}

We can see that the first two n-grams are the same, but the last is not. Using the coefficient:

$A \cup B$= Total distinct n-grams=4, $A \cap B$= total matched n-grams=2
Jaccard's Coefficient=2/4 =.5

As Jaccards' coefficient ranges on a scale 0 to 1 [no matches - all matched], we can also think of it as a percentage similarity, so our example above can be represented as 50%.

```
def float JaccardIndex(tokenList1, tokenList2){
   def intersectionSize = tokenList1.intersect(tokenList2).size()
   def unionSize = (tokenList1+tokenList2).unique().size()
   return  intersectionSize/unionSize
}
```
Table 4: Groovy's powerful list methods allows us to succinctly express Jaccard's Index

Tolerance

Varying the n-gram representation affects the 'tolerance' level of the comparison. That is, a comparison using 4-gram segments is less strict than comparisons using 5-gram segments. However, larger n-grams also have the undesirable property of not being sensitive to re-arranged code. The key then is to strike a balance. In testing, we found 5-grams to be a good tradeoff between performance and specificity. For simplicity, the rest of the paper will continue using 4-grams in the examples.

Performance

The performance of the program was assessed based on the average running time over 5 runs to check all files in a folder against all files in another folder (Table 5).

	Analysis Folder	Standards folder
Files	45	55
Lines of Code	6208	8550

Table 5: Breakdown of our test folders

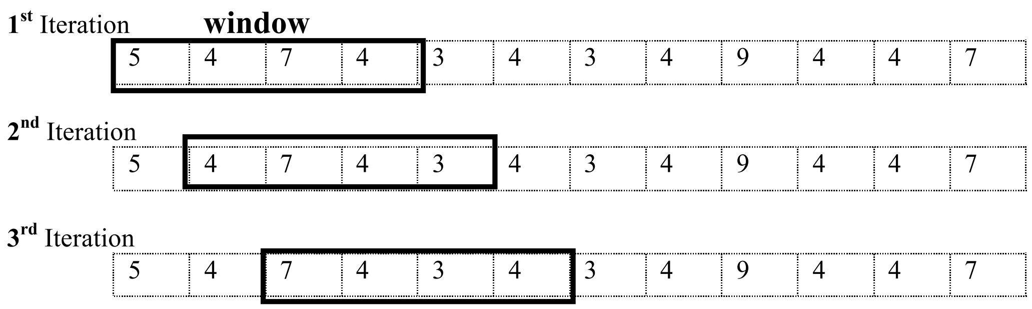
Using this implementation, the code took an average of 5.56 minutes to run. The performance bottleneck was found to be occurring in the calculation of Jaccard's Index. As the entire list of n-grams in a file was being loaded into memory before being manipulated, the performance was dependent on the size of this list.

An additional issue with the approach so far, is that due to the way we create the n-grams, we lose a lot of information on the context as we are only comparing n-gram partitions. More worryingly, simply placing one extra token at the start of a program, changes all the n-grams of that file.

Take for example, the n-grams we created earlier:

{5, 4, 7, 4} {3, 4, 3, 4} {9, 4, 4, 7}

We could have represented this with a sliding window of k (box below) producing the following k-grams



…

k-grams:
{5, 4, 7, 4} {4, 7, 4, 3} {7, 4, 3 4} {….}

This would certainly preserve more information, and make the program more flexible to changes at the start of the file. However, we now need to store much more data, so the performance is consequently also impacted.

On our performance test, this change brought the average running time to a whoopping 15.88 minutes.

Back to the drawing board

To improve the performance, we need to try reducing the space required to store the k-grams. We can start by applying a hash function.

Hash function

A hash function is an algorithm used to map a large dataset of variable lengths to shorter fixed lengths. Consider the following function (using a base of 10 for illustration):

$$\mathbf{h(k)=(k[0]10^4+ k[1]10^3+ k[2]10^2 + k[3]10^1 + k[4]1^0) \bmod p}$$

This means for each k-gram we can multiply by its base, raised to the power of the k-gram size minus the index + 1.

For our earlier example:

$$\mathbf{\{5, 4, 7, 4\} = \{5x10^3 + 4x10^2 + 7x10^1 +4x10^0\} = 5474}$$

The p in our function is a large prime number. Using the prime modulus of a number will allow us to reduce the size even further. However, it does mean that collisions can occur when more than one number maps to the same hash value. Therefore a large prime is recommended to reduce the probability of possible collisions.

Using the prime 37, we can now reduce our number to 35 (**5474 mod 37)**.

The effect of taking the modulus on storage is much more apparent for large k-grams or larger bases where the hash value can grow dramatically resulting in a sparsely populated hash.

Rolling hash function

A nice additional attribute of this hash function is that we don't have to compute the entire hash for each subsequent k-gram. Instead, we can use the first digit of the hash to help calculate the next hash using the following algorithm.

1. Remove the first digit (5474 becomes 474)
2. Multiply by the base (4740)
3. Add the next digit (4744)

More formally:

$$\mathbf{h(S_{i+1}) = [(h(S_i) - (10^4 * \text{first digit of } S_i)) * 10 + \text{next digit after } S_i] \bmod m}$$

We now only need to look at 2 elements, rather than k elements to compute the next hash.

Performance revisited

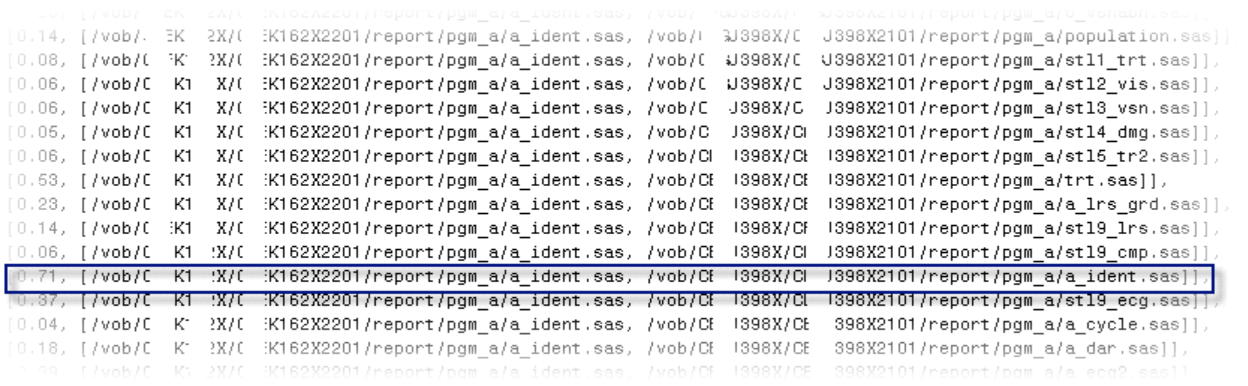
Recalling earlier, our code took 15.88 minutes to run. So how has our new hashing functions affected the performance?

Running the same test, on hashing alone, substantially decreased the time take to 12.86 **seconds**. A further improvement was observed using the Rolling hash function which completed the test in 11.93 **seconds**.

Discussion

Returning to our initial goals of our paper, we now have the tools required to answer our questions. To determine whether the standards are being used, we can check whether the highest scoring match of a program returns the corresponding standard program file.

In Figure 1, we can see the a_ident.sas program is highest scoring match is a 71% match with the standard programs a_ident. If we were to observe a highest scoring match that isn't the corresponding standard program, this can mean either: the program is not standard or the standard needs to be updated.

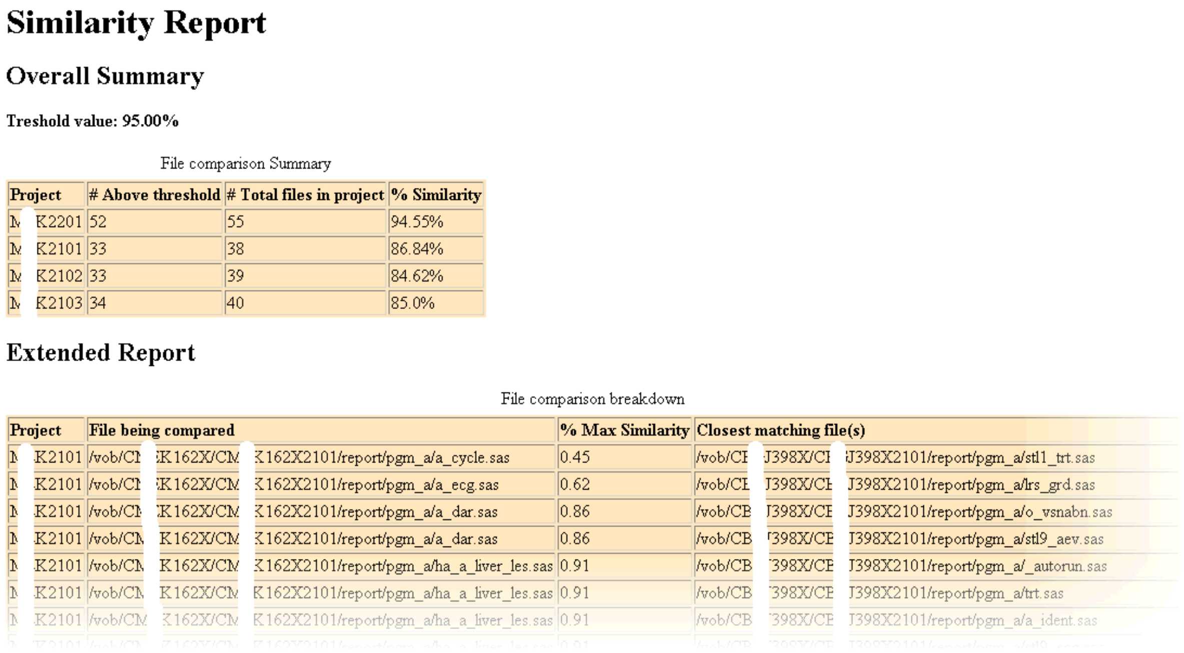


```
[0.14, [/vob/. EK  2X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/I  GJ398X/C   J398X2101/report/pgm_a/population.sas]],
[0.08, [/vob/(  K'  2X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/(  J398X/C   J398X2101/report/pgm_a/stl1_trt.sas]],
[0.06, [/vob/C  K1   X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/(  J398X/C   J398X2101/report/pgm_a/stl2_vis.sas]],
[0.06, [/vob/C  K1   X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/C  J398X/C   J398X2101/report/pgm_a/stl3_vsn.sas]],
[0.05, [/vob/C  K1   X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/C   J398X/CI  J398X2101/report/pgm_a/stl4_dmg.sas]],
[0.06, [/vob/C  K1   X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/CI  I398X/Ct  I398X2101/report/pgm_a/stl5_tr2.sas]],
[0.53, [/vob/C  K1   X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/CE  I398X/CE  I398X2101/report/pgm_a/trt.sas]],
[0.23, [/vob/C  K1   X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/CE  I398X/CE  I398X2101/report/pgm_a/a_lrs_grd.sas]],
[0.14, [/vob/(  :K1  X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/CE  I398X/CF  I398X2101/report/pgm_a/stl9_lrs.sas]],
[0.06, [/vob/C  K1  :X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/CE  I398X/CI  J398X2101/report/pgm_a/stl9_cmp.sas]],
[0.71, [/vob/C  K1  :X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/CE  I398X/CI  J398X2101/report/pgm_a/a_ident.sas]],
[0.37, [/vob/C  K1  :X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/CE  I398X/CI  I398X2101/report/pgm_a/stl9_ecg.sas]],
[0.04, [/vob/C  K'  2X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/CE  I398X/CE   398X2101/report/pgm_a/a_cycle.sas]],
[0.18, [/vob/C  K'  2X/(  :K162X2201/report/pgm_a/a_ident.sas, /vob/CE  I398X/CE   398X2101/report/pgm_a/a_dar.sas]],
```

Fig 1: Log of all comparisons made with match score

For the results to be useful, we also need to present this data differently. A high level project summary can help identify projects where the standards aren't being used. Additionally a low level breakdown of files compared with their best match will allow us to identify where standard programs could benefit from updating.

Groovy provides a convenient way to generate html dynamically through its MarkupBuilder class. So we can use this to create the report required (Fig 2).



Similarity Report

Overall Summary

Treshold value: 95.00%

File comparison Summary

Project	# Above threshold	# Total files in project	% Similarity
M K2201	52	55	94.55%
M K2101	33	38	86.84%
M K2102	33	39	84.62%
M K2103	34	40	85.0%

Extended Report

File comparison breakdown

Project	File being compared	% Max Similarity	Closest matching file(s)
N K2101	/vob/CI :K162X/CM K162X2101/report/pgm_a/a_cycle.sas	0.45	/vob/CI J398X/CI J398X2101/report/pgm_a/stl1_trt.sas
N K2101	/vob/CI :K162X/CM K162X2101/report/pgm_a/a_ecg.sas	0.62	/vob/CI J398X/CI J398X2101/report/pgm_a/lrs_grd.sas
N K2101	/vob/CI :K162X/CM K162X2101/report/pgm_a/a_dar.sas	0.86	/vob/CB J398X/CE J398X2101/report/pgm_a/o_vsnabn.sas
N K2101	/vob/CN K162X/CM K162X2101/report/pgm_a/a_dar.sas	0.86	/vob/CB '398X/CE J398X2101/report/pgm_a/stl9_aev.sas
N K2101	/vob/CN K162X/CM K162X2101/report/pgm_a/ha_a_liver_les.sas	0.91	/vob/CB '398X/CE J398X2101/report/pgm_a/_autorun.sas
N K2101	/vob/CN K162X/CM K162X2101/report/pgm_a/ha_a_liver_les.sas	0.91	/vob/CB '398X/CE J398X2101/report/pgm_a/trt.sas
N K2101	/vob/CM K162X/CM K162X2101/report/pgm_a/ha_a_liver_les.sas	0.91	/vob/CB '398X/CE J398X2101/report/pgm_a/a_ident.sas

Fig 2: Similarity report produced with a threshold value of 95%

Conclusion

To summarize, we started with the goal of quantifying the use of standard programs using Plagiarism Detection Techniques. We examined and implemented different techniques based on the limitations that we observed to produce a summary that can be used for monitoring our standard programs. With a program level review, we can identify standard programs that could or should be modernized. Finally, with a project level summary, we can identify those projects that are using a lot of non-standard programs, and see if we can encourage them to plagiarize a little more.

Acknowledgements

With thanks to Christelle Navarro, Nassim Sleiman, Samia Kabi, Sebastien Jolivet, Mathias Lebreton , Alexandra Emsellem and Dikey Ghung.

References

1. Roy, Chanchal Kumar;Cordy, James R. (September 26, 2007)."A Survey on Software Clone Detection Research". School of Computing, Queen's University, Canada.

2. Khurram Zeeshan Haider, Tabassam Nawaz, Sami ud Din, Ali Javed “Efficient Source Code Plagiarism Identification Based on Greedy String Tilling” IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.12, December 2010

3. A dynamic language for the Java platform http://groovy.codehaus.org/

4. Chanchal Kumar Roy and James R. Cordy “A Survey on Software Clone Detection Research”, http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf

5. Pornchai Kusonpalalert and Worawut Nawairrigulchai “Computer Engineering Project Progress Report on Source Code Plagiarism Detector” http://cpe.kmutt.ac.th/wiki/images/3/3e/SCPD_Progress_Report. pdf

6. Rolling Hash (Rabin-Karp Algorithm), 6.006 Intro to Algorithms 2006 courses.csail.mit.edu/6.006/spring11/rec/rec06.pdf

7. Robert Sedgewick and Kevin Wayne. Rabin Karp, Java program http://algs4.cs.princeton.edu/53substring/RabinKarp.java.html

8. Saul Schleimer et al. Winnowing: Local Algorithms for Document Fingerprinting www.math.uic.edu/~saul/Maths/winnowing.pdf

9. Okiemute Omuta. Electronic source code plagiarism detection, Computer Engineering Department,European University of LefkeLefke, North Cyprus: http://eul.academia.edu/kheme/Papers/353119/Electronic_Source_Code_Plagiarism_Detection

Appendix

```
proc groovy ;
submit ;

class SimpleTokenizer {
   def tokenListAb = []
   //def tokenList =[];

   def baseKeywords = ['libname','label','infile','filename','set',
      'file','by', 'proc','run','do','end','while','until','if',
      'else','cards','datalines','quit','data']//expand to suit need

   def macroKeywords = ['%abort','%copy','%do','%until','%while',
      '%end','%global','%goto','%if','%then','%else','%input',
      '%let','%local','%macro','%mend','%put','%return','%symdel',
      '%syscall','%sysexec','%syslput','%sysrput','%window']

   //First break into top level groups - string literal,
   //identifier,specials, numerics and whitespace
def re ='''(?si)(?:"[^"]*"|'[^']*')|%?[a-z_0-9]+|\\p{Punct}|\\d+|\\s+|.*'''

   def handleStatement(def statement){
      def scanner = statement =~ re
      scanner.each{match ->handleToken match}}

   def handleToken(def token) {
      //println 'Tokens '+ token
      switch(token) {
         // String literal can span multiple lines
         case ~/(?s)^\s*(?:\'|\").*$/:
            //tokenList << 'Literal'
            tokenListAb << 1
            break
         // Numbers
         case ~/^\d+$/:
            tokenListAb << 2
            //tokenList << 'Number'
            break
         // Identifiers
         case ~/^[A-Za-z_0-9]+$/:
            if (token.toLowerCase() in baseKeywords){
                //tokenList << 'bKeyword'
                tokenListAb << 3}
            else {//tokenList << 'bIdentifier'
                  tokenListAb << 4}
            break
         // Macro Identifiers
         case ~/^\%[a-zA-Z0-9_]+$/:
            if (token.toLowerCase() in macroKeywords){
              //tokenList << 'mKeyword'
              tokenListAb <<5}
            else {//tokenList << 'mIdentifier'
                 tokenListAb << 6;}
            break
         // Operator
         case ~/^[\+\-\/=]+$/:
            //tokenList << 'Operator'
            tokenListAb << 7;
            break
         // Separator
         case ~/^[(),]+$/:
            //tokenList << 'Separator'
            tokenListAb << 8;
            break
         // Punctuation
         case ~/^[^A-Za-z0-9_\s]+$/:
            //tokenList << 'Punctuation'
            tokenListAb << 9;
            break
         // Spaces or null
         case ~/^\s*$/:
            // do nothing
            break
         default:
            println 'Unhandled token ' + token
            break
         }
      }
   }


class RollingHash{
   static int base = 9    //Total token types used
   static int window = 5  //Window to compare
   static long P = 1009   //Define prime to use
   def hashedGrams

   def RollingHash(list) {
      def N = list.size()
      hashedGrams=[:]

      // precompute base^(window-1) % P for use in removing leading digit
      def long RM = 1;
      (window-1).times{RM = ( base * RM) % P }

      if (N<=window) {def long curHash = initHash(list,window); return}
      def long curHash = initHash(list[0..window-1], window);

      (window..N-1).each{
         // Remove leading digit, add trailing digit to compute new hash.
         curHash = (curHash + P - RM * list[it-window] % P) % P
         curHash = (curHash * base + list[it]) % P
         hashedGrams[curHash] = (hashedGrams[curHash] ?: 0) + 1
       }
   }

   // Calculate Initial hash value
   def initHash(list, window) {
      def h = 0;
      (0..window-1).each{h = (base * h + list[it])% P }
      return h
    }
}




//Calculate Jaccards similarity coefficient and convert to percentage
//Defined as size of intersect of two files divided by size of union
def float JaccardIndex(tokenList1, tokenList2){
   def intersectionSize = tokenList1.intersect(tokenList2).size()
   def unionSize = (tokenList1+tokenList2).unique().size()
   return  ((unionSize>0)? intersectionSize/unionSize:0)
}


// Parse a single file
def parseFile(filename){
   //Load entire file into memory
   def rawString = new File(filename).getText()

   //Remove C-style multiline comment
   def removeCComments = rawString.replaceAll('(?s)/\\*.*?\\*/','')

   //Break into SAS statements
   def statements = removeCComments.tokenize(';')

   //Pass each SAS statement to the tokenizer
   def tokenizer = new SimpleTokenizer()
   statements.each{statement ->
                   if  (statement =~ /(?s)^\s*\%?\*.*$/){}
                   else {tokenizer.handleStatement statement }
                   }
   //Split into k-grams and compute the hash function for each
   def h = new RollingHash(tokenizer.tokenListAb)

   //Create a list of only the k-grams
   def nGramList = h.hashedGrams.collect{key, value -> key}

   return  nGramList
}


// Proces a folder
def processDirectory(dirpath){
   def dict =[:]
   new File(dirpath).eachFileMatch(PROG_PATTERN){
      dict["${it}"] = parseFile("${it}") }
   return dict
}


// Write out the HTML report
import groovy.xml.MarkupBuilder
def HtmlWriter(summary, outpath){
   new FileWriter("${outpath}").withWriter{writer ->
      def xhtml = new MarkupBuilder(writer)
      xhtml.html{
         body(leftmargin:50){
            h1 "Similarity Report"
            h2 "Overall Summary"
            h4 "Treshold value: ${THRESHOLD*100}%"
            table(id:'Summary',
style:'border:1pxsolid;text-align:left;background-color: #FFE3BF',border:1){
               caption('File comparison Summary',size:20)
               tr{ th('Project'); th('# Above threshold');
                   th('# Total files in project'); th('% Similarity');}

               summary.each{proj, files ->
                  def above = 0
                  files.each { f-> def s= f[0][0];
                     (s>THRESHOLD)? above++ :above }
                  def total = files.size()
                  tr{td(proj); td(above);td(total);
                     td("${new Double(above/total*100).round(2)}%")}}
               }
            br{}

            h2 "Extended Report"
            table(id:'Breakdown',
style:'border:1pxsolid;text-align:left;background-color: #FFE3BF',border:1){
               caption('File comparison breakdown')
               tr{ th('Project'); th( 'File being compared');
                   th('% Max Similarity');
                   th('Closest matching file(s)', colspan:3)}

               summary.keySet().sort().each(){proj ->
                  def sorted=summary[proj].sort{it[0][0]}
                  sorted.each{elem-> elem.each{ sect ->
                     tr{ td(proj); td(sect[1][0]);td(sect[0]);td(sect[1][1])}
                 //println sect
                     }
                  }
               }
            }
         }
      }
   }
}
```

```
//////////////////////////////////////////////////
//Main calling section
//////////////////////////////////////////////////

PROG_PATTERN = ~/.*\.sas/
THRESHOLD = 0.8


// Store all standard programs
standardProgs = processDirectory(' path of standard program here')

dirpaths = ['Trial A':'path of trial A here…',
            'Trial B':'path of trial B here.',
            'Trial C':'path of trial C here …',
            'Trial D':'path of trial D here….']

def dirScores =[:]
dirpaths.each{proj, dirpath ->
   def fileScores = []
   new File(dirpath).eachFileMatch(PROG_PATTERN){checkpgm ->
      def cnGrams = parseFile("${checkpgm}")
      def allScored = standardProgs.collect{ stdpgm, snGrams ->
         [JaccardIndex(cnGrams,snGrams).round(2),[checkpgm,stdpgm]]}

      //Keep only the highest scored matches for each file
      //println allScored;
      maxs = allScored.findAll{it[0]==allScored.max{it[0]}[0]}
      fileScores << maxs
   }
   dirScores[(proj)] = fileScores
   //println dirScores
}

HtmlWriter(dirScores,"output path here… ")


endsubmit;
quit;
```