PharmaSUG 2019 - Paper AP-018

Ensuring Programming Integrity with Python: Dynamic Code Plagiarism Detection

Michael Stackhouse, Covance Inc.;

ABSTRACT

Integrity in your team's programming is of the utmost importance, not only when preparing for a regulatory submission, but simply to ensure the quality of the analysis throughout your drug's development. A common validation technique used throughout the industry is double programming - where two programmers work independently to obtain identical results. But how do you guarantee independence? And what if you suspect programming independence was violated? Linux tools like diff may not be sufficient to identify harmful similarities. This paper will explore a Python based tool that can bring these issues to light. The tool is dynamic enough to locate similarities at any point in your programs, and can allow flexibility for minor syntactic changes, such as dataset name changes or reordering of statements. All results are gathered into easily reviewable files to help ensure that your team's work can uphold the integrity and reputation that you rely on.

INTRODUCTION

Analyzing text data can be a complex topic. Certain aspects of it are very simple. For example, consider the following two sentences.

1) `I have two dogs.`
2) `I have one cat.`

For starters, simply by looking at the text you can tell that the sentences are not equivalent. Furthermore, most programming languages can very easily tell you that the sentences are not equal. In a single line we can see this in Python:

```
In [1]: "I have two dogs." == "I have one cat."

Out[1]: False
```

Linux systems offer us tools like diff, which can do more advanced analysis than simply telling us that the two sentences are different. diff can show you all the lines in a file that are not equivalent. Take the following two documents as an example:

DOCUMENT1.TXT

```
I have two dogs.
I have one cat.
My dog's names are Dash and Kara.
My cat's name is Reggie.
```

DOCUMENT2.TXT

```
I have two dogs.
They like to play.
I have one cat.

My dog's names are Dash and Kara.
My cat's name is Reggie.
```

From the Linux command line:

```
>>> diff document1.txt document2.txt
1a2
> They like to play.
2a4
```

This gives some very useful information. First, we can see that the documents are quite similar. There are only two lines of text that are different; one is the text *"They like to play"*, and the other is a blank line that was inserted. But this value starts to diminish as the documents get larger and the text contains more differences. Consider a SAS program. One program can easily be more than 1500 lines of code.

Furthermore, *diff* has another significant deficiency. Comparisons are line by line – so even if two documents are quite similar, the output can become quite extensive. Consider the following:

DOCUMENT1.TXT

```
I have two dogs.
I have one cat.
My dog's names are Dash and Kara.
My cat's name is Reggie.
```

DOCUMENT3.TXT

```
I have one cat.
I have two dogs.
My cat's name is Reggie.
My dog's names are Dash and Kara.
```

From the Linux command line:

```
>>> diff document1.txt document3.txt
1d0
< I have two dogs.
3c2
< My dog's names are Dash and Kara.
---
> I have two dogs.
4a4
> My dog's names are Dash and Kara.
```

In this small example, it is still straightforward to see that the lines were just reordered. When this example is scaled up to a few hundred lines in a SAS program it becomes more complex to see that a block of code at line 300 in Program 1 is at line 450 in Program 2. This all makes sense considering that the tool is named *diff* and not *sim* – as it is locating the differences in a text file and not the similarities.

So what do you do when you're looking for similarities in a file and not the differences? What if lines do not have to be identical, but can be reordered and slightly altered? When you are evaluating programming independence, these situations may present themselves. A lazy copy will be identical, but a more thought out one could reorder statements, change variable names, change *!=* to ^=, or more. All these modifications can trick a tool like *diff.*

These issues and more will be addressed throughout the course of this paper, using Python as the language of choice. The reasons for choosing Python will become apparent from the examples, but some major reasons are Python's powerful string operations, simple file handling, and most of all – the Scikit-learn libraries.

As you may imagine, developing a tool like this has gone through a few iterations. The success of one method leads to new deficiencies that need to be addressed – and there is still room for improvement with where I am now. That being said, the journey to get to where I am has been valuable, and throughout the course of this paper the different iterations are discussed, along with their strengths and weaknesses. Some of the critical topics covered include:

- Distance measures of text
- Multiprocessing
- Text vectorization

At the end of this paper, you should have a high-level understanding of how text data can be analyzed to determine similarity. Better yet, you will see some of the pitfalls that I have encountered while developing this tool, and approaches that I have taken to make the tool more efficient and useful. All these concepts have been put together to assemble Covance DETECT (Dynamic Evaluation of Technical Conformance Transgressions).

Note: Please see Appendix 1 for instructions on the installation of Python versions and libraries in this paper.

VERSION 1

Development of DETECT started with a simple question: How can I easily check to see if code was copied between a production and validation program. That is the dream, is it not? To run a simple utility, see a list of all the programs you would like to compare, and immediately know which of them are problematic? I hate to disappoint you, but for reasons covered later, it is not that simple – but this was the motivation behind version number one.

This brings us to an important concept: measuring the similarity of text. The first attempt to build DETECT applied this on a document level – meaning, all of the text is taken from document 1, all of the text is taken from document 2, and the similarity between the two documents is measured. This is done using some sort of distance metric. Those of you with backgrounds in statistics should understand this concept well, and there are many metrics and methods of measuring distance available, but the application must be well understood when choosing a metric.

The method used in version 1 implemented a token sort ration, using a Python library named fuzzywuzzy. Here is a simple example of how a token sort ratio works, and why it is beneficial in this context:

```
In [1]: from fuzzywuzzy import fuzz
In [2]: fuzz.token_sort_ratio("I have a dog", "I have a cat")
Out[2]: 75
```

Great! In the sentence above, there are four words, of which three are similar – so this is a simple and intuitive scoring of the two sentences' similarity. With three out of four words matching, the similarity score is 75. So the 75 can be interpreted similar to a percentage. There are further benefits to this metric as well:

```
In [3]: fuzz.token_sort_ratio("I have a dog", "DOG A HAVE I")
Out[3]: 100
```

This shows two important points. First, the metrics already takes into consideration case sensitivity. As SAS syntax is not case sensitive, casing is not of much concern. Second, and more importantly, the token sort ratio sees past the order of text, which is something that a traditional diff does not do. In a SAS program, if a=b means the same thing as if b=a, so this also allows for similarity to be evaluated on text that is not ordered identically as well.

So now we have something fairly powerful – a clean, interpretable metric that tells you how similar two programs are - except for two major problems:

1) SAS syntax is inherently similar

2) It does not tell you *where* the programs are similar.

For the first point, while the first tests of this method performed well, after further testing it was apparent that at the document level, the scores meant nothing. A programming language does not have the variety of a spoken language, so a SAS program will inherently have many of the same "words" as another SAS program, and thus the scores can easily be inflated even if the programs aren't truly "similar".

For the second point, an overall score is convenient and a very helpful thing to have, but it did not solve the problem of finding *where* the similarities exist – and this is the true goal that needs to be accomplished. Even if you know that a program has problems, you need to identify where those problems are. Furthermore – and this is a very important point that needs to be remembered throughout the course of this paper – **a single line can still be a major problem.** Code copying is not isolated to one programmer copying another programmer's entire program. A single line can be copied, and that can still be extremely problematic.

All these problems lead to the second iteration of DETECT.

VERSION 2

The major problem that version 2 set out to address was the location of similarities. The core question is still there – how do I identify text that is similar? Rather than completely shift paradigms, this required a refocus of attention. In order to identify location, the program needs to be split into separate segments – and then the similarity of those segments is checked, rather than the similarity of the whole document. The easiest way to do this is to look at the program by lines. Why lines? Firstly, it is very simple in Python (and most other languages, really) to read in a program by line. In Python, you can read in a program and immediately split it into a list:

```
with open('program.sas', 'r') as f:
    text_in_list = f.readlines()
```

This results in an iterable list of lines in the text file – which is very useful for what we are trying to do. Second, splitting a program by lines allows you to easily track the location of any matches and trace it back to the original file. For example, if line 20 in program1.sas matches line 50 in program2.sas, you can very easily navigate to these locations in the original SAS programs.

That leads us to the challenge; find the location of any line in program2.sas that match any line in program1.sas. Let's start with some pseudocode and walk through it step by step.

```
# Start a dictionary variable
list_of_matches = {}

# Loop over all lines in Program 1[1]
for line_a in Program 1:
    # Loop over all lines in Program 2[2]
    for line_b in Program 2:
        # Check the similarity[3]
        similarity = distance(line_a,line_b)
        # If the lines are similar enough, track the number[4]
        if similarity > threshold:
            list_of_matches[line number of line_a].append(line number of
line_b)
```

While this is a simplification, this is the crux of the algorithm still being used in DETECT. Walking through the algorithm (following the superscripts above):

1) Program 1 is used as the base – so this is the outside of the loop. The resulting dictionary that is produced will be the line number in Program 1, with all its matching line numbers in Program 2.

2) This is a brute force approach. While not quite efficient, I believe it is required to find all possible matches. A match can be anywhere – any line in Program 1 to any line in Program 2. Therefore, we compare all lines in Program 1 to all lines in Program 2. The efficiency issues with this approach are addressed in later versions.
3) For each comparison, the similarity needs to be checked. This is measured by some distance metric. In Version 2, the token sort ratio was still used.
4) Once the similarity is established, if it is passed a certain threshold, the line number in Program 2 is tracked. The threshold is something that will vary based on the distance metric being used. In Version 2, a threshold of 95 was set. This is what allows for some flexibility beyond just the ordering of words in the line, as after sorting, there can be some variation in the exact text used.

A visual representation of one iteration of this loop can be thought of as follows:

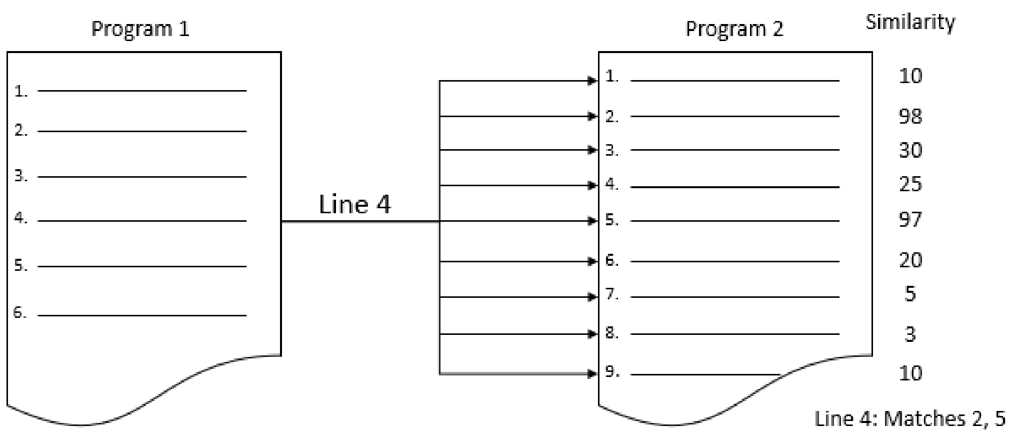

Figure 1.

Another point worth noting is that some lines will inherently have a large number of matches. For example, SAS programs will naturally have many short lines, such as *run;*, *end;*, and *quit;*. To avoid these more innocent matches, in Version 2 lines shorter than 7 characters were simply filtered out using this small modification to the algorithm:

```
 # Start a dictionary variable
list_of_matches = {}

# Loop over all lines in Program 1
for line_a in Program 1:
    # Loop over all lines in Program 2
    for line_b in Program 2
        # Check the similarity
        similarity = distance(line_a,line_b)
       if len(line_a) > 7:
            # If the lines are similar enough, track the number[4]
            if similarity > threshold:
                list_of_matches[line number of line_a].append(line number of
line_b)
```

With all this information collected, the next challenge becomes the presentation of these findings. Once the matches are available, both Program 1 and Program 2 need to be reviewed to check what the matches are. Version 2 attempted to consolidate some of this information to make it more easily reviewable. To do this, the line matches found, and Program 1 were combined. In the resulting output of a program, the line numbers of the matched lines in Program 2 are prefixed to each line in Program 1. The resulting output is as follows:

```
Similar to: [131]---------------------> %macro _rename(str, var_out);
Similar to: [132]--------------------->    %do i=1 %to
%sysfunc(countw(&str));
Similar to: [133]--------------------->     %let item=%scan(&str,&i);
Similar to: [134]--------------------->       %let
item2=%scan(&var_out,&i);
--------------------------------------->       &item = &item2
--------------------------------------->   %end;
```

Now, when reviewing Program 1, you can immediately know where to look within Program 2 to find a suspicious match.

While this made a significant improvement over Version 1, some issues still existed:

- The run time of longer programs can sometimes exceed 4 minutes, so it could easily take over 30 minutes to finish all of the programs for a study.
- The output is still fairly extensive. The output files provide much more useful information, but there is still quite a lot of noise to sift through to find true issues – so it still does not address the core issue of finding a needle in the haystack.

These two problems were the main motivation behind the development of Version 3.

VERSION 3

Version 3 introduced some exciting concepts. The goal in this version was to speed up the algorithm, without a loss of information, and offer a high-level summary that can give some insight as to which programs may be the most problematic.

While the execution time of the program does not have a direct impact on the quality of the results, it can certainly be a nuisance. Waiting half an hour for the program to complete can be tedious and running the utility across a vast number of programs could simply be implausible given time constraints. Unfortunately, in this case reducing the complexity of the algorithm is not easily achievable. If Program 1 is 1000 lines, and Program 2 is 1000 lines, then to evaluate every possible pair of matches, there must be 1,000,000 comparisons. But the resource constraint in this scenario is real time; memory and processing power realistically are not issues. So the question becomes, how can the algorithm be accelerated without reducing the amount of work being done?

One slight nuisance with Python is something called the Global Interpreter Lock. The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time (Real Python). In simple words, you can only do one thing at a time, even though most modern processers should be able to do about four (a dual core processor has four threads). But like almost anything else in Python, there are ways you can circumvent this issue.

The multiprocessing library in Python allows you to split a task between multiple interpreters – effectively allowing you to make use of more than one thread to complete the task at hand. Consider the following example:

```
import multiprocessing as mp

def f(x):[1]
    return x

if __name__ == '__main__':[5]
    with mp.Pool(4) as p:[2]
        pools = [p.apply_async(f, args=(x,)) for x in range(4)][3]

        for x in pools:[4]
```

```
            try:
                results.append(x.get())
            except:
                results = [x.get()]

    print(results)
    print("Done")
```

Which yields the results:

```
[0, 1, 2, 3]
Done
```

In this example, a very simple function, that just returns its input value, is called for the values 0 through 3. The mechanics around how this is done are what makes it interesting. Following the superscripts above:

1. Function *f* is defined, which accepts input *x* and returns that exact value.
2. Separate pools are set – these are the separate Python instances that will be opened. Remember that the GIL restricts to one interpreter. To circumvent this, multiple instances are opened so multiple interpreters can be used.
3. The *apply_async* function submits the function call, *f*, to one of the pools. The *args* parameter takes the input to the function – but the arguments must include any object that will be utilized by the function, since a new instance of Python is initialized.
4. The interesting part of *apply_async* is that the separate pools are run asynchronously. Since they're not required to run in any specific order, the parent instance of Python does not automatically retrieve the results. Instead, the results need to be requested from each of the separate pools. Otherwise, the program will not pause and will attempt to continue executing.
5. Lastly, the line *if __name__ == '__main__'* is a crucial aspect of using multiprocessing in Python, and also helps explain how this process works. To obtain the information needed to run each individual subprocess, Python reads in the parent program file. The variable __name__ is a built-in variable that contains the name of the module being executed. The original program that is executed will have *__name__ = '__main__'*. For each of the subprocesses, *__name__ = '__mp_main__'*. If this line is left out, Python will infinitely keep resubmitting subprocesses.

While in this example multiprocessing is far from necessary, the important concept is that different functions could be evaluated simultaneously. When the amount of work required increases to a process that can take multiple minutes, the value gained increases dramatically.

The time taken to evaluate the program can be substantially cut down by breaking the processing into smaller chunks. The goal is to check every line in Program 1 against every line in Program 2, but using a single process, you can only check one line in program one at a time. Using multiprocessing, you can check *multiple* lines in Program 1 against Program 2 at one time, slicing the time taken to evaluate the whole program significantly. Simply section off program one into equivalent chunks by as many threads as you have and evaluate each of those chunks simultaneously. Think of it like this:

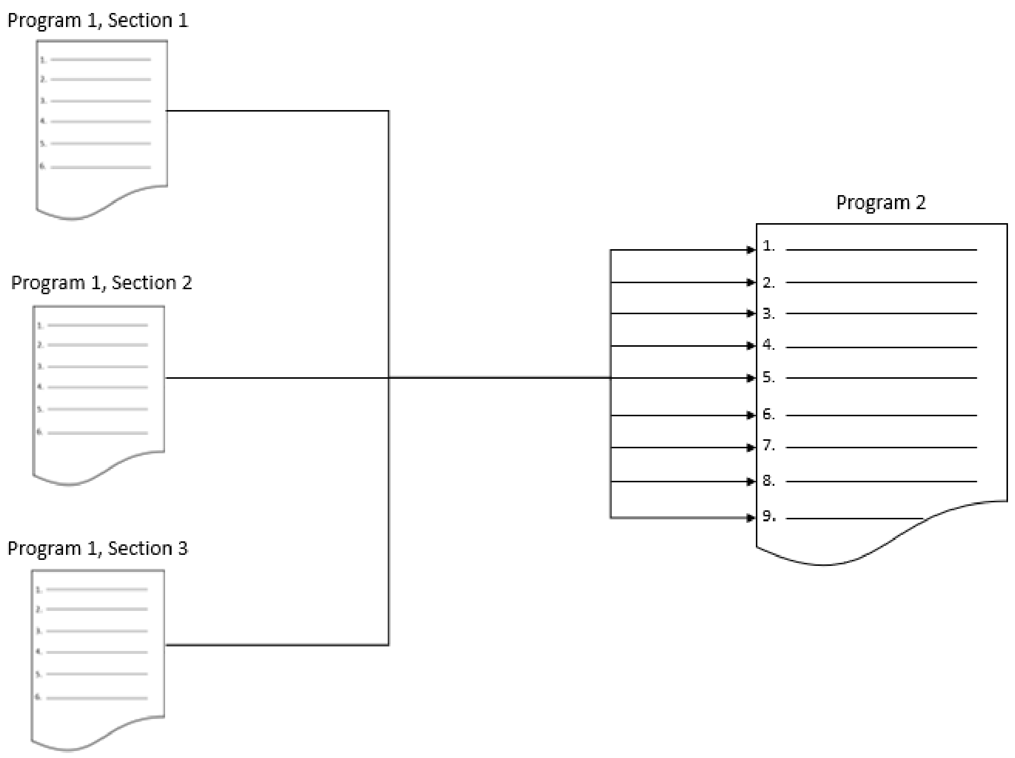

Figure 2.

While this is not quite an increase in efficiency, the real time benefits are significant. The graph below shows the number of lines in a program on the X axis, and the time in seconds on the Y axis. While there is diminishing marginal utility of adding a concurrent process, the benefits of the additional threads are quite clear.

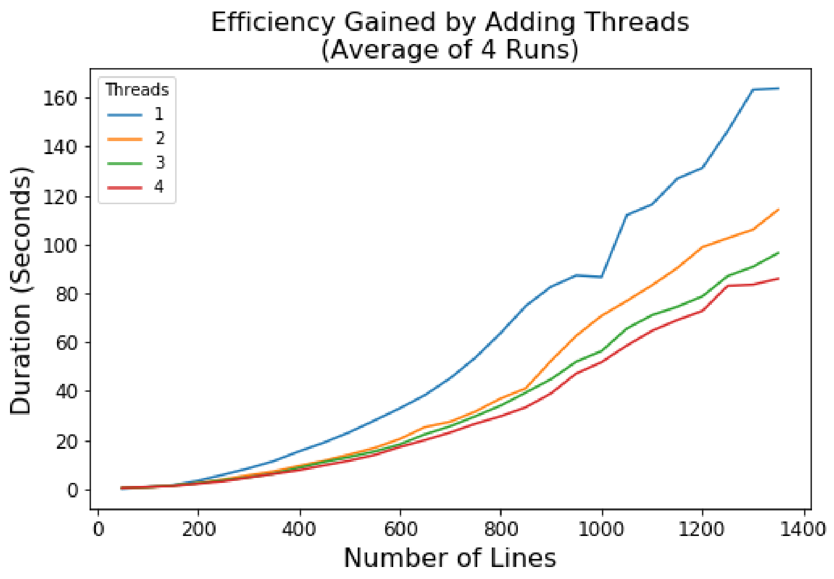

Figure 3.

The second goal of Version 3 was to offer a better high-level summary to get a quick idea of which programs may be problematic. This started the idea of tracking information about each program that is being evaluated. Version 3 tracked three main data points:

1. Total lines in Program 1
2. Matched lines in Program 1
3. Ratio of matched lines to total

While this is not much information, these are useful details to consider when evaluating which programs may be problematic. The number of matched lines is the key piece of information – the more matched lines, the more likely it is there is an issue in the program that needs to be addressed. The ratio gives you the portion of the program that matches its counterpart, so a higher ratio helps you understand the perspective of the number of matched lines. Finally, the total number of lines in Program 1 gives you a scaling factor. For example, it is much easier to inflate the ratio of matched lines to total in a 100-line program than a 1000-line program. For review, Version 3 presented this information in a bar chart.

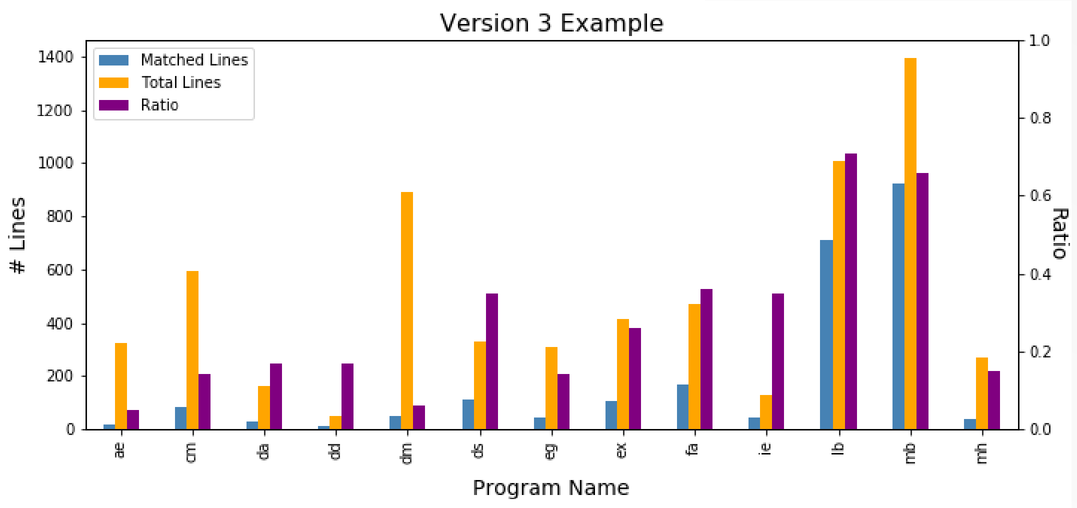

Figure 4.

While Version 3 made some significant improvements over Version 2, there was still much to be desired. First, review of the line matches still involved hunting down the matched line numbers in Program 2. Second, while the speed was substantially increased, there is still much to be desired when a program can take over two minutes to finish comparing. Lastly, though the summary review graph can give you a quick impression of major problems, it does little to help find a single line that can cause a problem. Version 4 addresses two of these problems and sets the groundwork for the third.

VERSION 4

Version 4 is where some major changes come into play. While the core concept of the algorithm remains, essentially everything else was rebuilt from the ground up. And it is here that we will need to make some detours to discuss concepts important to how this version works.

This version delivers some substantial improvements, including:

- A different choice of distance metric
- Dramatically enhanced speed
- More detailed metadata per program
- Improved output delivery format

- Improved foundation for further development

Before digging in, an important concept needs to be discussed that explains some of the improvements in Version 4 – text vectorization.

TEXT VECTORIZATION

Natural Language Processing (NLP) is a very powerful tool, but this sort mathematical analysis cannot be performed directly on the strings in which text is typically captured. First, the text needs to be converted into numbers. The most straightforward way that this can be explained is to simply capture the frequency of the words appearing in a string. Consider the following example:

```
string1 = "I have a dog and his name is Dash"
string2 = "I have a cat and his name is Reggie"
```

Here we have two sentences with some minor differences. But to represent these strings as vectors, you can transform them with the frequency of each word as a token. As a visual example:

	I	HAVE	A	DOG	CAT	AND	HIS	NAME	IS	DASH	REGGIE
string1	1	1	1	1	0	1	1	1	1	1	0
string2	1	1	1	0	1	1	1	1	1	0	1

Table 1.

Now each string has converted nicely into an array. This step also takes care of a few other concerns discussed earlier when looking at the token sort ratio. First, the ordering of the words is now irrelevant. When the text matrix is created, each word is assigned to a column, so sorting is now unnecessary. Furthermore, if there were differences in casing – this too would have been circumvented during this process. Of course, if the order of words and casing are relevant to your specific use case, changes to the process can be made to take those factors into consideration.

Now that the text is a numeric array, different operations and analysis can be performed. This brings us to the next point in version 4, which is the change to the distance metric.

COSINE SIMILARITY

In Version 4, I chose to use cosine similarity instead of the token sort ratio. However, this change has more to do with the programming options available in Python than an issue with the token sort ratio metric itself. The results between the two metrics are fairly similar, which aids in interpretation of the similarities between older versions and Version 4. The similar resulting scores also make sense, as token sort ratio and cosine similarity are similarly designed. Both neglect the order of the text, but both also take into consideration the frequency of words in a string, rather than just the set of words. Cosine similarity offers another benefit, which is that it is also computationally efficient. After normalizing the count vectors to have a length of 1, the cosine similarity can be calculated with a dot product. The derivation is as follows:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

(Gupta, S.)

Figure 5.

The code below shows the steps necessary to find the cosine similarity of two strings in Python, which also outlines the steps taken to vectorize text in Python.

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import CountVectorizer
from fuzzywuzzy import fuzz

def get_cosine_similarity(s1,s2):
    ''' Detremines the cosine similarity between two strings'''

    # Combine string1 and string2 into one string to vectorize
    corp = ' '.join([s1,s2])

    # Create the vectorizer object. CountVectorizer does not consider 
single
    # character words by default, so override the tokenizer
    vectorizer = CountVectorizer(token_pattern = r"(?u)\b\w+\b")

    # Fit the vectorizer - this step determines all of the columns that
    # will be necessary
    vectorizer.fit([corp])

    # Create each of the individual text vectors - this step takes the 
string and
    # makes the actual array of text frequency counts
    s1_vector = vectorizer.transform([s1])
    s2_vector = vectorizer.transform([s2])

    # Now we can take the actual cosine similariy and return the score.
    # The [0][0] just pulls the number out of a nested array.
    return cosine_similarity(s1_vector, s2_vector)[0][0]

# Set the strings to be compared
string1 = "I have a dog and his name is Dash"
string2 = "I have a cat and his name is Reggie"

# Display the results for both cosine similiary and token_sort_ratio
print("Cosine similarity = {}".format(get_cosine_similarity(string1, 
string2)))
print("Token sort ratio = {}".format(fuzz.token_sort_ratio(string1, 
string2)))

Results:
Cosine similarity = 0.7777777777777779
Token sort ratio = 76
```

SPEED IMPROVEMENTS

Why make this change if the resulting measurement is relatively the same? As a matter of fact, by following the steps detailed above the speed of the algorithm can be *dramatically* enhanced – by about 99%. A comparison that could take 2 minutes, even with multiprocessing, was reduced down to under 2 seconds in Version 4. As I mentioned, one factor here is the computational efficiency of deriving the cosine similarity, but the other much more relevant factor is the count vectorization steps.

In Versions 2 and 3, the tokenization of each text string had to be done at every comparison.   If you need to compare two 1000 line programs, each individual string would need to be tokenized 1000 times. This is 999 times too many. By vectorizing the text up front and creating your text frequency matrix, all that work is done in one step, so when the cosine similarity is taken, the original strings are ignored completely and instead the text frequency matrices are used. In short, a large deal of redundant computation is avoided – and it just happens to be the most computationally intensive steps.

While this is clearly the more efficient method of finding the matched lines, it is also more abstract. The algorithm used in Version 2 is more intuitive. Therefore, I felt that explanation of Version 1 through 3 were worthwhile to truly understanding the motivation and challenges of achieving this result.

REDESIGN

Version 4 was developed with an object-oriented design. Prior versions to this were implemented in a scripting framework, meaning that the utility simply runs from top to bottom of the program once executed – much like a SAS program. Redesigning the tool using and object-oriented design allows for much greater flexibility. Each pair of programs that are run through a comparison becomes a match object – which makes it very simple to track information about those match objects as attributes. Furthermore, different capabilities can easily be added to these match objects in the form of methods. This enhanced flexibility lays the ground work for the tool to grow even more without major rework.

More detailed metadata in Version 4 also proves very useful to offering greater insights. For each match object, Version 4 captures:

- At the line level
    - Original lines
    - Matched lines
    - Matched line similarities
    - Top matches
    - Short match filtered lines
    - Block indicator
    - Block ID
- At the summary level
    - Total Lines
    - Number of matched lines
    - Matched line percentage
    - Number of short match filtered lines
    - Short matched filtered line percentage
    - Number of blocks

A couple of these data points warrant further explanation. The short matched filtered lines are simply lines that are greater than 7 characters. In this version, I removed the initial filter of these lines and instead captured both the filtered and unfiltered lines. Having the shorter lines included in the total match percentage is worthwhile to identify if a program was completely copied. For example, including *all* lines would show you something more like 95%, where the short match filter may be closer to 75%.

The block indicator and block ID are two of the most valuable additions in Version 4. A block is a multiple line segment of code in Program 1 that has a matching multiple line segments of code in Program 2 – but the blocks do *not* have to be in the same locations. Consider the following two programs:

PROGRAM 1

```
data x;
    set y (where=(var1 ^= var2) keep=var1 var2);
run;

proc sort data=x;
    by var1 var2;
run;

data z;
    merge x (in=a) y (in=b);
    by var1 var2;
    if x;
run;
```

PROGRAM 2

```
proc sql;
    create table x as
        select var1, var2 from y order by var1, var2;
quit;

data z;
    merge x (in=a) y (in=b);
    by var1 var2;
    if x;
run;
```

Despite the fact that the two sections of code are not at the same location in each program, the bolded blocks above would be considered matching blocks. The programming logic in DETECT considers the matched line numbers from Program 1 to Program 2, and if sequential lines were found in both programs, a block is identified. The block indicator is a flag that allows for simple filtering to only blocks of matched code. The block ID is an identifier for each individual block, and also serves as a simple counter for the number of blocks in the match object.

After blocks are identified, another important attribute to the match object can be found – the top matching line. The top matching is either a) the line with the highest cosine similarity, or b) a line that was found to be within the corresponding block to the current line in Program 1. The block is prioritized over the highest cosine similarity because if code was copied between the programs, it is likely that the full block was taken, rather than an assortment of different lines.

The top match also serves another important purpose, which is used in formulating the output report for Version 4. The top matched line number is taken from Program 2 to pull the text of the actual line in Program 2. The lines are then paired together so they can be displayed side by side.

OUTPUT FORMAT

None of this information is truly worthwhile unless it is easily reviewable. Version 4 attempted to improve the quality of the review documents. To do this, all the output was put into excel. This is where all the additional metadata can be reported. There are two main components to this excel workbook:

- The Summary Tab
    - This is where aggregate level attributes are displayed about each program
- The Program Match Tabs

- This contains a full print out of each line in Program 1 paired with its top match to Program 2.
- Additional line level attributes are also presented.

Note: Examples of each of these tabs can be seen in Appendix 2.

The use of excel allows for a much more favorable review experience. First, with additional metadata such as block indicators, or short match filters, unnecessary noise can be filtered out to focus on lines that may be more problematic. Additionally, presenting in excel allows for simple presentation of each line in Program 1 matched with its top match to Program 2, eliminating the need for the user to have multiple documents open at one time. Now, the user can visually compare the lines that could be problematic right next to each other.

Even with all these enhancements and additional data points captured, there is still one problem that has yet to be fixed. What about the single line that could be a problem? Though not yet implemented, this one of the best aspects of Version 4. The output excel format serves a secondary purpose – data capture of human review.

WHERE TO GO FROM HERE

Version 4 is not where this tool will stop. While the current version can find the matched lines and offer information for you to judge whether or not the lines are problematic, it cannot make that judgement for you. That being said, all of the important metadata surrounding the program summaries or match objects are presented in the excel file. Once reviewed, each problematic line can be flagged, and each program with issues can be marked directly in the excel files – in fact the columns for this review have been provided.

Once a number of these human reviews have been completed, these files can then be read in and assembled into a labeled dataset. Having these data will allow for the application of machine learning algorithms to start predicting whether or not a matched line of code is truly problematic. With enough data, an additional flag could be created to indicate whether or not a line is likely and issue. This is where the problem of finding that single line could truly be solved – as with a trustworthy and reliable filter, you can truly find the signal through the noise.

While machine learning will allow DETECT to flag a line as problematic, it is important to note that it will *not* be able to make a definitive judgement as to whether the intent was malicious. There are simply too many possibilities as to why a line of code matched between two programs. For example:

- Specs may have been written with code in the definitions
- Code may have been provided in the SAP
- Code may have taken from completely separate programs developed by the same person, who used the same programming logic (i.e. ISO8601 date formatting)

And there are many more possible situations. This is to not say that it is ok that the lines match identically; this could still very well be a problem, but a line identified to be problematic does not necessarily say anything about intent.

CONCLUSION

One line of copied code can wreak havoc on your programming team. It can compromise your study's analysis, and it can erode trust with your sponsor. Ensuring integrity in your deliveries is critically important on multiple levels. Covance's DETECT tool allows us to review our programming teams' work and easily ensure integrity in the products we deliver. DETECT employs some natural language processing techniques to efficiently review two programs and identify similar lines of code between the two programs, and further work is being done to clearly and succinctly identify the lines that are most problematic.

APPENDIX 1: PYTHON INSTALLATION DETAILS

Python Version I Use:	Python 3.6.4 :: Anaconda, Inc.
Anaconda Download Link:	https://www.anaconda.com/download/
Downloading Other Python Versions:	http://docs.anaconda.com/anaconda/user-guide/faq/#how-do-i-get-the-latest-anaconda-with-python-3-5
Pandas Version:	0.22.0
Numpy Version:	1.14.0
Scikit-learn Version:	0.19.1
Installing FuzzyWuzzy	Execute *pip install fuzzywuzzy* from your command line after installing Python. You can ignore the import warnings about *python-Levenshtein* as it requires the installation of Microsoft Visual C++ Build Tools and will require administrative privileges. FuzzyWuzzy still works without that library, you just get a warning on import.

APPENDIX 2: VERSION 4 EXCEL OUTPUT EXAMPLES

SUMMARY PAGE

Production Program	Production Program Path	Production Program Length	Validation Program	Validation Program Path	Validation Program Length
dm.sas	…	892	dm.sas	…	547
ae.sas	…	326	ae.sas	…	319
cm.sas	…	593	cm.sas	…	603
eg.sas	…	306	eg.sas	…	410

CONTINUED:

Total Matched Lines	Total Match %	Short-match Filtered Lines	Short-match Filtered %	Number of Matched Blocks
126	14.13%	54	6.05%	15
40	12.27%	18	5.52%	2
140	23.61%	90	15.18%	21
68	22.22%	44	14.38%	10

MATCH OBJECT TAB

Production Program	Matching Validation Line
/*%let keepvars = studyid domain usubjid egseq egrefid egtestcd egtest egcat */	/*%let keepvars = STUDYID DOMAIN USUBJID EGSEQ egrefid EGTESTCD EGTEST EGCAT */
/* egpos egorres egorresu egstresc egstresn egstresu egstat egreasnd egnam */	/* EGPOS EGORRES EGORRESU EGSTRESC EGSTRESN EGSTRESU EGSTAT EGREASND EGNAM EGLEAD EGMETHOD*/
%let keepvars = STUDYID DOMAIN USUBJID EGSEQ EGREFID EGTESTCD EGTEST EGCAT EGPOS EGORRES EGORRESU	%let keepvars = STUDYID DOMAIN USUBJID EGSEQ EGREFID EGTESTCD EGTEST EGCAT EGPOS EGORRES EGORRESU
EGSTRESC EGSTRESN EGSTRESU EGSTAT EGREASND EGNAM EGLEAD EGMETHOD EGBLFL EGEVAL VISITNUM VISIT	EGSTRESC EGSTRESN EGSTRESU EGSTAT EGREASND EGNAM EGLEAD EGMETHOD EGBLFL EGEVAL VISITNUM VISIT

CONTINUED:

Short-match Filter	Block Indicator	Block ID	All Matched Lines in Validation	Matched Lines Cosine Similarity
Y	Y	1	2	1.0
Y	Y	1	3	0.905
Y	Y	2	7	1.0
Y	Y	2	8	1.0

REFERENCES

Real Python. (2018, June 10). What is the Python Global Interpreter Lock (GIL)? – Real Python. Retrieved from https://realpython.com/python-gil/

Gupta, S. (2018, May 15). Overview of Text Similarity Metrics in Python – Towards Data Science. Retrieved from https://towardsdatascience.com/overview-of-text-similarity-metrics-3397c4601f50

CONTACT INFORMATION

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

Author Name	Michael Stackhouse
Company	Covance, Inc.
Address	4000 Centregreen Way
City / Postcode	Cary, NC 27513
Email:	Michael.Stackhouse@Covance.com
Web:	https://www.linkedin.com/in/michael-s-stackhouse/