THE WEAPON - THE USE OF KOLMOGOROV COMPLEXITY TO COMBAT PLAGIARISM IN THE CLASSROOM

George Sturm

Radio Logos, Pogradec, Albania

ABSTRACT

This paper discusses a unique usage of Kolmogorov Complexity that combats the bane of all teachers – plagiarism. The theoretical foundation of this tool is presented and illustrated using real-life data from the area of software programming.

KEYWORDS

Kolmogorov Complexity, plagiarism, cheating, cluster analysis.

1. INTRODUCTION

A simple definition of plagiarism is the use of another person's work as your own. It manifests itself in multiple fields - literary work, scientific research, university-level academics and software programming, just to name a few. It is a serious problem that is often equated with theft, fraud and stealing. The ancient saying "there is nothing new under the sun" is a good description for how long plagiarism has plagued society.

Plagiarism detection tools have been around since the existence of the problem. For sure, compared to today's options, early detection methods were archaic, time-consuming and limited in their scope of detection. Nevertheless, they served a purpose – to protect academic honesty. It goes without saying that many academic reputations have been ruined once plagiarism was revealed.

Today, it is standard procedure that prior to journal publication a scientific paper must first be checked for plagiarism. Many options are available to do this, some of which are free tools but with limited options while others are more comprehensive based upon various pricing plans. It is not stretching the truth to say that the field of plagiarism detection has grown into a large and competitive business.

This paper focuses on a narrow field of plagiarism, not that of publishing a scientific paper, but the detection of cheating in the classroom, that is, students copying from one another.

The motivation for this paper came about in my introductory class on Information Theory when I noticed that students would submit very similar looking source code for projects that were assigned, not as a team project, but to be completed individually. The same problem was noticed in the technical reports that were required from each student, that is, great similarities in the language content.

The question arose as how to solve this problem. As any teacher will testify, it is a time-consuming task to not only grade a project, but also to check it for plagiarism. A simpler and faster way needed to be developed, a method that would also teach the students about honesty and academic integrity. Seeing that Kolmogorov Complexity was one of many topics to be taught in the course, it was a natural way of teaching its concept and to solve the problem of plagiarism.

2. METHODS

Fisher's Information and Shannon Entropy are two common measures from Information Theory that are used to describe the "complexity/diversity" of a random variable. While these measures are challenging enough to explain to the students of an introductory course on Information Theory, Kolmogorov Complexity is far more difficult.

The reason for this is that Kolmogorov Complexity of an object (e.g. a character string) is not well-defined in that it lacks a precise mathematical formula. Its definition falls, rather, into more of a philosophical realm and is defined to be the length of the shortest computer program that outputs the object. A slightly different wording is that Kolmogorov Complexity is the minimal amount of computer resources needed to produce the object.

While the above definitions might be grammatically understandable, to a typical student in an introductory Information Theory course, they are very abstract and remote especially when it is added that Kolmogorov Complexity is incomputable in theory.

To bring Kolmogorov Complexity into an understandable and practical realm, the following teaching methodology was employed. First, the students would be assigned a software project with clear instructions. That is, each student was required to write their own software code, to produce a runnable program, and to not copy from any other student. A warning was also given that if a student was caught plagiarizing, then he would receive zero points for the project. Of course, as in any typical Computer Science class, plagiarism is a great temptation.

Second, the student projects would be, of course, graded to see if they worked appropriately or not. However, the real goal of this project was to invoke a plagiarism detection technique that was based on Kolmogorov Complexity. To the surprise of the students, all plagiarism became as obvious as the noon-day sun. When asked how this detection was so easily done, the answer led to a natural discussion on Kolmogorov Complexity. I must add that this technique was so effective that it quickly became known, in their own words, as "Professor Sturm's weapon" and the incidents of plagiarism in future semesters dropped drastically in my classes.

Specifically, in the initial semester where this methodology was employed, the class consisted of twenty students (D=20). By using the R language [1], they were assigned a project to read in the plain text shown in Table 1 [2], invoke Huffman coding, then the RSA encryption algorithm using randomly selected prime numbers to produce the Private and Public keys. The project ended by the student invoking the reverse steps to re-produce the actual plain text. It is noteworthy that most of the twenty programs performed perfectly and that most students were to receive a perfect grade, that is, until the "weapon" was introduced.

Table 1. The plain text.

Introduction to Information Theory Dr. George Sturm, Ph.D. Without authorization from George Sturm no reproduction of this book in any form (photocopy, digital reproduction, etc.) is permitted. Forward

Unknown to the students, a simple plagiarism detection technique was developed using Kolmogorov Complexity. Although, in theory it is incomputable, there are empirical methods to approximate it. Zenil [3] describes several alternatives, but a discussion of these techniques was beyond the scope of an introductory class on Information Theory. Seeking a more understandable approximation, we decided to follow the advice of Bloem [4, page 346] where he says "To construct practical applications (ie. runnable computer programs), the most common approach is to take one of these platonic, incomputable functions, derived from Kolmogorov complexity (K), and to approximate it by swapping K out for a computable compressor like GZIP." That is, we used the R-language base function memCompress(filename, type="gzip") as an approximation of Kolmogorov Complexity.

The algorithm (shown in the R language) for the plagiarism detection technique goes as follows. First (see Table 2), the software programs of all twenty students are read in. See [5] for the data. To protect the privacy of the students, their names are simply S01, S02, . . . , S20. The variable 'files1' is the full address of the file whereas 'junk1' is simply the shortened file name (e.g. S01.R). The variable 'doc' is the scanned software code and 'name' is the name of the student (e.g. S01).

Table 2. Algorithm -Part 1.

```
rm(list = ls())

files1=list.files(path = "C:\\Plagiarism",
       pattern = ".R", all.files = FALSE,
       full.names = TRUE, recursive = TRUE,
       ignore.case = FALSE, include.dirs = FALSE, no.. = FALSE)
junk1=list.files(path = "C:\\Plagiarism",
      pattern = ".R", all.files = FALSE,
      full.names = FALSE, recursive = TRUE,
      ignore.case = FALSE, include.dirs = FALSE, no.. = FALSE)
D = length(files1)

doc=rep(NA,D)
names=rep(NA,D)
cnt=0
for (i in 1:D){
  docc = scan(files1[i],what="character",sep="")
  cnt=cnt+1
  names[cnt]=substr(junk1[i],1,nchar(junk1[i])-2)
  doc[cnt]=paste(docc,collapse="")
}
```

Next, using gzip, we approximate the Kolmogorov Complexity for each of the student programs. This variable is called 'KS_approx.' The code is shown in Table 3 and the values in Table 4. It must be noted that the variable 'txt.gz' is a vector of hexadecimal values. Thus, KS_approx is the length of this vector.

Table 3. Algorithm -Part 2

```
KC_approx=rep(NA,D)
for (i in 1:D){
  txt.gz <- memCompress(doc[i], type="gzip")
  KC_approx[i]=length(txt.gz)
}
KC_approx
```

Table 4. Approximate Kolmogorov Complexity values for individual students

Student	KC_approx	Student	KC_approx
S01	1682	S11	1257
S02	1564	S12	2020
S03	1279	S13	1325
S04	1407	S14	1395
S05	1846	S15	1969
S06	1313	S16	2555
S07	1500	S17	1846
S08	1605	S18	1645
S09	1430	S19	1096
S10	1800	S20	1430

The next step was to combine the students' source code in a pairwise fashion and to calculate the Kolmogorov Complexity for each pair. The idea is that if one student, say student S01, copied the source code from another, say S02, then the pairwise complexity value would be essentially that of student S02. Whereas if each wrote their own code, then the pairwise complexity score would be roughly the sum of the two complexity scores. The code for this is shown in Table 5 where a DxD matrix is generated.

Table 5. Algorithm -Part 3

```
KC_pairwise=matrix(rep(0,D*D),nrow=D, ncol=D)
string=matrix(rep(" ",D*D),nrow=D, ncol=D)
for (i in 1:D){
for (j in 1:D){
  string[i,j]=paste(doc[i],doc[j],collapse="", sep="")
  txt.gz <- memCompress(string[i,j], type="gzip")
  KC_pairwise[i,j]=length(txt.gz)
}
}
KC_pairwise
```

The final calculation was to create the "adjusted" Kolmogorov Complexity score (KC) for each pairwise document. This was simply the pairwise complexity score KC_pairwise(i,j) minus the score the complexity score of document i, that is, KC_approx(i) where i,j =1,2,. . .,D. The code is shown in Table 6 and the KC matrix in Table 7.

Table 6. Algorithm -Part 4

```
KC=matrix(rep(0,D*D),nrow=D, ncol=D)
for (i in 1:D){
for (j in 1:D){
  KC[i,j]=KC_pairwise[i,j]-KC_approx[i]
}
}
KC
```

Table 7. The pairwise adjusted Kolmogorov Complexity scores (KC)

(i,j)	1	2	3	4	5	6	7	8	. .	17	18	19	20
1	61	1304	857	1064	1333	735	1255	1356		1333	1375	874	1140
2	1452	65	1039	932	1386	1070	1279	1370		1386	1245	860	984
3	1273	1306	47	1085	1539	653	1303	1383		1539	1420	880	1142
4	1369	1096	975	48	1316	983	1250	1357		1316	1189	880	950
5	1187	1108	967	873	63	976	1251	1337		63	1048	852	719
6	1145	1313	649	1081	1535	44	1299	1394		1535	1430	882	1165
7	1464	1336	1074	1136	1590	1098	58	1363		1590	1394	900	1197
8	1447	1320	1056	1138	1579	1098	1247	70	. .	1579	1387	876	1195
. .								. .	. .	. .			
17	1187	1108	967	873	63	976	1251	1337	. .	63	1048	852	719
18	1434	1153	1049	930	1229	1086	1245	1352		1229	55	891	868
19	1467	1314	1060	1183	1611	1072	1296	1380		1611	1451	42	1211
20	1422	1136	1003	932	1148	1040	1265	1365		1148	1104	885	50

The first notable thing about this DxD matrix is that the diagonal values, although small, are not zero. This might be surprising to the reader since the algorithm for a diagonal element is the Kolmogorov Complexity value for the document appended to itself minus the complexity value of the single document. Thus, a reasonable expectation is that this value should be zero. However, probably due to the information created due to the appending of the files, the value comes out slightly larger than zero.

The second notable item is that the matrix is not symmetrical. This is due to the fact that the equation for KC(i,j) is not symmetrical. That is, the equation KC(i,j) = KC_pairwise(i,j) – KC_approx(i) is dependent on the order of the joining of the two documents. However, the two sides of the diagonal matrix do reflect the two possible pairwise orderings of the documents.

The question now becomes – How do we use this KC matrix to determine if plagiarism was taking place? The answer is Cluster Analysis.

Cluster Analysis is a classical statistical technique that helps the researcher “reduce” or “simplify” the data into similar clusters. In our situation, Cluster Analysis will group the students into similar clusters based upon the KC matrix. We used the pvclust method [6] from the R software package version 4.2.1. [1]. Internally, the package uses the AU (Approximately Unbiased) probability value to construct these clusters. The idea is that if the AU value is greater than or equal to 95, then we can group students into a single cluster. The R code is shown in Table 8. Applying this technique to our data, a Cluster Dendrogram (see *Fig 1*) is produced which is a graphical method to assess which student source codes are similar.

Table 8. Algorithm -Part 5

```
library(pvclust)
A = matrix(t(KC), nrow=D, ncol = D)
dimnames(A) = list( names, names)

A.pv <- pvclust(A, nboot=1000)

plot(A.pv, cex=0.8, cex.pv=0.7, col.pv=c(1,0,8))

ask.bak <- par()$ask
par(ask=TRUE)
## highlight clusters with high au p-values
pvrect(A.pv, alpha=0.95)
```

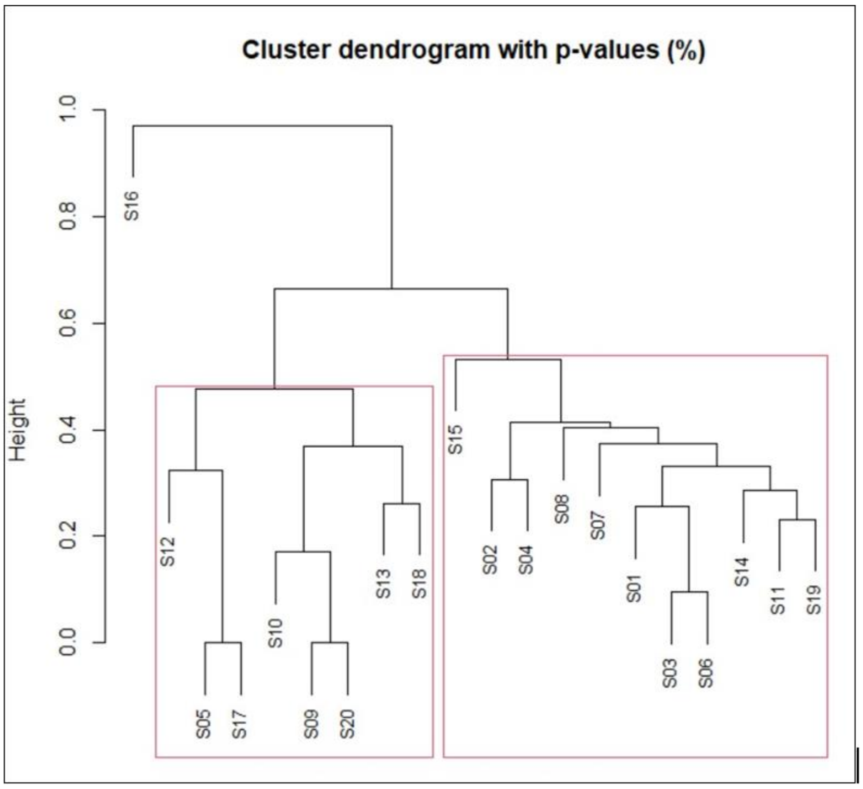

Fig 1. Dendrogram for plagiarized source code

The Dendrogram shows that the students can be grouped into three (3) clusters (see *Table 9*).

Table 9. Clusters of students

Cluster 1	Student 16
Cluster 2	Students 05,09,10,12,13,17,18,20
Cluster 3	Students 01,02,03,04,06,07,08,11,14,15,19

The Cluster Analysis clearly reveals that student 16 had the most unique source code, whereas the other students were at risk of collaboration in writing their code. These clusters proved to be of great practical help in that they provided direction as to which students needed to be carefully investigated.

For example, a careful hands-on examination in Cluster 2 started with students 05 and 17 which showed that their code was virtually identical. The same was true of students 09 and 20. The code of student 10 had great similarities with that of students 09 and 20. This investigation continued with the other students in Cluster 2.

Likewise, for Cluster 3. A physical examination of the source codes for students 03 and 06 revealed significant similarities. As one went up the cluster Dendrogram, the similarities, of course, decreased.

In the end, some students received zero points due to significant plagiarism while others were penalized less according to the detected scale of plagiarism.

3. RESULTS

The results were multi-faceted. First, the students all acknowledged that this methodology combined with the problem of plagiarism was very effective in helping them understand the concept of Kolmogorov Complexity.

Second, it took the application of this methodology on several succeeding projects before the reality set in that plagiarism does not pay. At first, it was a bit humorous to the students that their plagiarism was so easily exposed. However, after receiving zero points on several projects in a row, the students began to understand that “Professor Sturm has a weapon that we need to respect.” Thereafter, the incidents of plagiarism took a dramatic drop.

Third, this effect continued into future semesters. Although, the word had spread throughout the student population that such a weapon existed, it turned out that the students really didn’t believe it, that is, until after the first project was assigned and many received a zero. Thereafter, the problem of plagiarism was effectively eliminated for the remainder of the semester.

Finally, the students were often assigned technical reports to write. Like with source code, there was the temptation of plagiarism in their Word documents. Once again, the application of the above methodology effectively eliminated the problem.

4. CONCLUSION

This simple usage of Kolmogorov Complexity proved to be very effective in combating plagiarism, not only in software programming, but also in composition writing. It also served an effective way to teach the abstract concept of Kolmogorov Complexity.

In addition, this methodology is very easy to implement in any programming language and is free to the user. That is, the user does not have to pay any expensive user fees.

Although our methodology is conceptually simple, this does not mean that there is no need for improvement. Areas for future research include how to integrate into this methodology ‘jumps’ in paragraph quality in the field of composition writing. That is, plagiarism might be the inclusion of an unreferenced paragraph from a professional academician. If so, then the quality of the student's work might suddenly take a 'jump' in lexical richness, grammar correctness, style, and word length. Perhaps Kolmogorov Complexity or another diversity measure could be used to detect such 'jumps.'

REFERENCES

[1] R Core Team (2014). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/

[2] Sturm, G.W. (2017). *Introduction to Information Theory*. Pyongyang University of Science and Technology, DPRK.

[3] Zenil H. A Review of Methods for Estimating Algorithmic Complexity: Options, Challenges, and New Directions. Entropy (Basel). 2020 May 30;22(6):612. doi: 10.3390/e22060612. PMID: 33286384; PMCID: PMC7517143.

[4] Bloem P., Mota F., de Rooij S., Antunes L., Adriaans P. A safe approximation for Kolmogorov complexity; Proceedings of the International Conference on Algorithmic Learning Theory; Bled, Slovenia. 8–10 October 2014; pp. 336–350..

[5] Sturm, G.W. (2025). Link to plagiarism files. https://1drv.ms/f/c/8b328b74ef334b0b/EqudekSmckJFqKxdL32y7l8BLc9h6POmOFpjHfsV_fVM8A?e=dCFwnQ

[6] Ryota, S., Terada, Y., Shimodaira, H. (2019) pvclust: Hierarchical Clustering with P-Values via Multiscale Bootstrap. https://CRAN.R-project.org/package=pvclust.

AUTHOR

George Sturm was born in the USA but lives in Albania. He received his Ph.D. from The Ohio State University in 1979. His research interests include statistics, mathematics, electric engineering, and radio broadcasting.