Plagiarism Detection Approaches for Simple Introductory Programming Assignments

Sven Strickroth [1]

Abstract: Learning to program is often perceived as hard by students and some students try to cheat. Plagiarisms are reported to be a huge problem particularly for summative-like assignments (e.g., crediting courses or bonus points). It is important to fight plagiarisms from early on – even for simple assignments. Especially for larger courses tool support is required. This paper provides an overview of features for commonly used plagiarism detection tools, discusses how these can be integrated into existing assessment systems, and how their results relate to each other for two data sets of quite simple assignments. Additionally, these specialized tools are compared with a simple Levenshtein distance approach. The paper also outlines limits on very simple assignments.

Keywords: automatic assessment; plagiarism detection; programming education

1 Introduction

At universities programming is often taught in lectures with accompanying homework assignments where learners are asked to write (small) programs from scratch, implement interfaces, or to expand existing programs. However, many learners face severe difficulties in turning the theoretical knowledge into code [La18]. A common approach is to use special tools that handle the submission of the learner's solutions and automatically provide feedback [KJH18; SP17]. However, not all learners solve the assignments on their own or even hand in plagiarized solutions. Plagiarism and other forms of cheating are often reported as a huge issue [La18]. This is particularly true in scenarios in which assignments are graded and used e.g. for crediting a course or for bonus points in the exam.

In smaller courses where one lecturer or tutor can assess all submissions, plagiarisms can be detected manually. However, when dealing with large courses with more than 100 students and multiple tutors grading subgroups of the cohort, it is not effectively possible to detect plagiarism that occur in different groups. Particularly for these scenarios specialized plagiarism detection tools are required.

From the experience of the author, it is very important to establish an environment of good scientific conduct and to fight plagiarism from early on. Therefore, it is necessary to have detection mechanisms that can handle also simple assignments. However, experience has shown that most of the plagiarism detection tools seem to be developed for more advanced

[1] LMU München, Institut für Informatik, Oettingenstraße 67, 80538 München sven.strickroth@ifi.lmu.de

doi:10.18420/abp2021-6

assignments or even programming projects. The problem is that for quite simple assignments lots of false positives are reported that need to be inspected manually (cf. [MPH16]).

The contribution of this paper is to show preliminary research results on comparing existing tools regarding the effort to integrate these into existing assessment systems and the results produced by different plagiarism detection approaches on quite simple assignments written in Java. This should also help to select proper thresholds for the tools. The existing specialized tools are also compared with a simple Levenshtein approach because it is hypothesized that this works better for more simple assignments regarding the number of reported plagiarisms.

2 Related Research

[NJK19] performed a comprehensive systematic literature review on programming plagiarisms in academia. Their research includes definitions of academic plagiarisms, used detection approaches, reported learner's obfuscation methods, evaluation methods, and used data sets. In research there are several plagiarism detection methods developed. As mentioned in [NJK19] the "top-five" most-mentioned tools are JPlag, MOSS, Sherlock-Warwick, Plaggie, and SIM-Grune. These methods are also sometimes directly integrated into assessment tools [SP17]. It is salient that a lot of tools are not available for use.

There are also several papers systematically comparing these methods. The majority of the research [CLS21; HRV11; Lu14; Ma14] seems to focus on the robustness of plagiarism detection evasion with "simulated undergraduate plagiarism" and/or a comparison on features and performance. A notable exception is the comparison of [MPH16] where a real data set was used to analyze the tools regarding false positives and false negatives. However, a problem there is that it is not clearly stated when a found potential plagiarism (as the detection tools output a percentage) is counted as a found one. Also, the used versions are not specified frequently. Hence, there is a research gap for comparing plagiarism detection methods on real-world data sets of introductory programming courses.

3 Plagiarism Detection Methods

The tools used for comparison in this paper are JPlag, SIM, Plaggie, Sherlock-Sydney, and Moss. The rationale for the selection of these tools was the availability of the tools. Additionally, a very simple approach based on the Levenshtein distance is also analyzed because a hypothesis is that this simple approach is particularly helpful for simple assignments. For a short comparison see Tab. 1. With the exception of Moss all tools can be used offline and are available as source code. Only JPlag and Plaggie seem to rely on fully-fledged parsers for their token-based approach and both might ignore submissions they were not able to parse. Plaggie seems to be most critique here: Usage of broken Java code (e.g., missing "}") or newer Java features such as the diamond operator with Generics (introduced in

Java 1.7) or lambda expressions (Java 1.8) make the parser fail – JPlag seems to be a bit more resistant and also doesn't seem to complain for newer language features but, still, a significant number of parse errors could be observed in practice [cf. AM19]. Other tools do not seem to have issues with unparseable Java code or unknown syntactic constructs. All tools support submissions to be organized in separate folders – one for each submission and the outputs are per cent values as an indicator for the degree of similarity. Except for Moss all reported results are symmetric. Apart from JPlag and Moss (and the GATE implementation of the Levenshtein approach) the tools do not seem to be actively maintained.

Tool	Release	Algorithm	Java support	source available	offline
JPlag[a]	2.12.1 (2019)	Greedy String Tiling (token-based)	yes ($\leq$ 9)	yes (GPLv3)	yes
Levenshtein distance	(GATE)[b]	Levenshtein distance	text-files	yes	yes
Moss[c]	n/a[g]	winnowing (token-based)	yes[g], incorrect syntax ok	no	no
Plaggie[d]	2006	Greedy String Tiling (token-based)	yes ($\leq$ 6), only correct syntax	yes (GPL)	yes
Sherlock-Sydney[e]	n/a[g]	winnowing (token-based)	text-files	yes (public domain)	yes
SIM[f]	3.0.2 (2017)	token-based	yes[g], incorrect syntax ok	yes (BSD)	yes

[a] `https://github.com/jplag/jplag`
[b] [SOP11], `https://github.com/csware/si/tree/5618eeefdf0395db86046ec1f5e7a5e0e8d45338`
[c] `https://theory.stanford.edu/~aiken/moss/`
[d] `https://www.cs.hut.fi/Software/Plaggie/`
[e] `https://github.com/diogocabral/sherlock`
[f] `https://dickgrune.com/Programs/similarity_tester/`
[g] no version specified

Tab. 1: The evaluated approaches/tools

JPlag is written in Java and can be downloaded as a JAR file. Besides Java several other languages are supported and boilerplate code can be specified. It can be used from the command-line and provides the results in form of two CSV files (average and maximum similarity) but can also be used by directly via an API. For matches HTML files are generated. A special feature is the clustering of similar submissions. By default, reports only show the top 20 clusters/similar submissions (i.e., default threshold). **Levenshtein distance** is a generic algorithm, here the implementation from the GATE system was used. The similarity is returned as per cent values (1 minus the number of deletions/additions divided by then maximum length) when comparing two files of two submissions. Before comparing, a normalization can be applied: deleting Java comments, lower casing all characters, and converting/reducing tabs, spaces, new lines into a single space. **Moss** is only available as a web-based service and can handle multiple languages. There are different submission clients available. The result is a link to a web page where a full table containing the per cent values as well as separate pages for each match can be obtained – this interface is not optimal for

automatic processing. Also, Moss was not available multiple times for several hours during the tests. Moss seems to have a threshold of 250 reported pairs. **Plaggie** is written in Java. It can be downloaded as source code and can only handle Java code. Plaggie can be used from CLI as well as other Java programs. HTML files for matches can be generated and boilerplate code can be ignored. The detection works on a per submission basis. Plaggie only handles syntactically correct Java 1.6 source code. By default, it reports all results with $\geq 50\,\%$ similarity. **Sherlock-Sydney** is implemented in C and handles generic text documents. It works on a per file basis and reports all per cent values to STDOUT in a CSV style. By default, it reports all results with a similarity $\geq 20\,\%$. **SIM** is written in C and binaries for "MS-DOS" are available (the Makefile does not work at least on current *nix systems). It can handle different languages and works on a per file basis, however, files from the same submission are not compared (see SIM documentation) and boilerplate code cannot be specified. SIM outputs similarities to STDOUT, optionally as per cent values.

4 The Data Set & Method

The focus in this research lay on quite simple assignments written in Java (1.8). Assignment 1 is from an Introductory Programming Course for learners studying business administration, assignment 2 is from a first semester programming course for computer scientists. **Assignment 1:** Calculation task (LoC: 25, main method, variables, calculations, 209 submissions, 60 known plagiarisms with 63 pairs): For an installment payment the difference of paying cash and total price incl. interests as well as the zero-difference case should be calculated and printed out with rounded numbers (2 decimals). Input (arg[]): cash price, cash down, rate per month and period in months. **Assignment 2:** main method containing a single "if else-if else"-construct or nested if-statements and returning given strings based on "arg[0]" to distinguish 3 cases. Approx. 20 lines of code, 616 submissions.

Special in this data set is that the design of the exercises of the first lecture allowed couples of learners to submit solutions and also included the explanation of the submitted solutions in presence of a tutor. Before the audition the tutors tried to detect plagiarisms (supported by Plaggie and the Levenshtein approach) and could, then, confront the learners – for the course explaining and being able to re-code a solution allowed them to get the points under certain conditions. For the second lecture, no plagiarisms are known (cf. next section).

For the evaluation all tools were used with default parameters despite that we tried to get all pairs of solutions that were not classified as completely different (i.e., threshold $\geq 1\,\%$). JPlag and Moss have hard-coded limits of 1000 and 250 results. Moss' asymmetric results are converted to symmetric ones by using the maximum of the two similarities. The found similarities are then used to calculate the number of reported results, the number of found known plagiarisms, the histograms for reported similarities, the inter-rater agreement (Cohen's κ) on the classification that a tuple of two submissions has a similarity $\geq 80\,\%$, and some descriptive statistics on the reported similarities. As a ground truth the known plagiarisms are used that were found by the tutors and confirmed by learner interviews.

5 Results

All tools needed less than 15 minutes to complete. The detection results for the assignments are shown in Tab. 2 and 3. LV means Levenshtein distance and LVN Levenshtein distance with the mentioned normalization. The columns are the number of results (≥ 1 %) that a tool reported, the number of results with the default threshold (cf. Section 3), the number of reported pairs with a similarity ≥ 80 %, the maximum reported similarity of a tool, and the mean as well as the median of all similarities (on the full data set, including all 0 % similarities). For the first assignment, the number of found known plagiarisms respective the maximum/mean per cent value on the found plagiarisms is reported in parenthesis. The data show for the first assignment that all tools detected all similarities in some way, however, the assigned per cent values seem to be very different when the mean and median values are compared. Here, JPlag, Plaggie and SIM have a median of ≥ 99. Particularly, Sherlock-Sydney stands out here, only detecting 24 plagiarisms with the default threshold and general low mean scores (22.3 for the known plagiarisms). Exemplary, Fig. 1 shows the histogram of reported similarity scores for assignment 1 and allows to get an impression on how many results are reported for different thresholds. For the second assignment Tab. 3 shows that most approaches report a high number of high similarities – even with a reported similarity score ≥ 80 %. Particularly, the simple Levenshtein distance is notable here with an average ≥ 60 % as well as the low numbers for JPlag and Moss due to the low hard-coded thresholds. Interestingly, also SIM has very low numbers compared to the others.

Tool	No. results	No. def. thr.	No. ≥ 80 %	Max %	Mean %	Median %
JPlag	1000 (63)	40 (26)	190 (59)	100 (100)	3.2 (93.8)	0 (100)
LV	21528 (63)	21528 (63)	8 (8)	100 (100)	25.8 (52.3)	0 (46)
LVN	21528 (63)	21528 (63)	43 (36)	100 (100)	36.4 (81.7)	0 (65)
Moss	250 (63)	250 (63)	17 (14)	99 (99)	0.6 (62)	0 (58)
Plaggie	7647 (63)	227 (53)	90 (46)	100 (100)	8 (80.6)	0 (100)
Sherlock	11217 (63)	174 (24)	4 (4)	100 (100)	2.9 (22.3)	0 (17)
SIM	317 (63)	317 (63)	31 (27)	100 (100)	0.6 (51.2)	0 (99)

Tab. 2: Reported similarities of the analyzed tools on assignment 1 (in parenthesis the numbers for found known plagiarisms); LV: Levenshtein distance; LVN: Levenshtein distance with normalization

Tab. 4 and 5 show the inter-rater agreements. A value $\leq .2$ is considered as "no", $.21 - .39$ as "minimal", $.4 - .59$ as "weak", $.6 - .79$ as "moderate", $.6 - .8$ as "strong" and above as "almost perfect" inter-rater agreement [Mc12]. For assignment 1 (Tab. 4) the agreements of Plaggie/JPlag and LV/Sherlock are "moderate", SIM/LVN and Moss/SIM are "weak". Others such as Sherlock/ JPlag&Plaggie are very low. The human classification seems to agree "moderately" with Plaggie and LVN. For assignment 2 (Tab. 5) most agreements are very low – notable exceptions are Plaggie/LVN (strong) as well as LVN/LV and Plaggie/LV (weak). In general, most tools do not seem to agree well on quite simple tasks for a ≥ 80 % plagiarism classification and, therefore, cannot be interchanged easily. A more in-depth analysis is necessary, also whether this is different for more complex assignments.

Tool	No. results	No. def. thr.	No. ≥ 80 %	Max %	Mean %	Median %
JPlag	1000	288	1000	100	0.5	0
LV	189420	189420	31355	100	61.9	2
LVN	189420	189420	79165	100	73	3
Moss	250	250	15	99	0.1	0
Plaggie	99326	75431	52209	100	38.5	0
Sherlock	164968	81883	1051	100	19.4	0
SIM	978	978	290	100	0.3	0

Tab. 3: Reported similarities of the analyzed tools on assignment 2

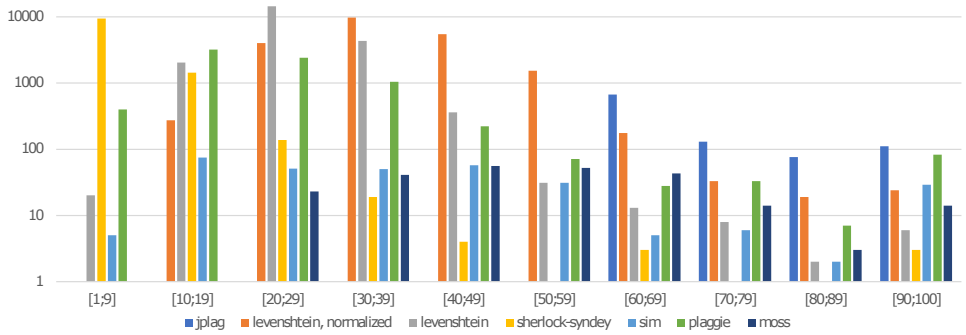

Fig. 1: Histogram of reported similarity per cent values for assignment 1

	Plaggie	Moss	SIM	Sherlock	LVN	LV	Human
JPlag	0.641	0.144	0.279	0.041	0.358	0.080	0.464
Plaggie		0.242	0.429	0.063	0.480	0.142	0.600
Moss			0.583	0.190	0.366	0.400	0.350
SIM				0.228	0.594	0.410	0.574
Sherlock					0.170	0.667	0.119
LVN						0.274	0.678
LV							0.225

Tab. 4: Inter-rater agreement on assignment 1 for ≥ 80 % classification

	Plaggie	MOSS	SIM	Sherlock	LVN	LV
JPlag	0.027	0.000	0.024	0.019	0.015	0.017
Plaggie		0.000	0.007	0.016	0.681	0.472
Moss			0.039	0.000	0.000	0.001
SIM				0.022	0.004	0.008
Sherlock					0.011	0.024
LVN						0.431

Tab. 5: Inter-rater agreement on assignment 2 for ≥ 80 % classification

6 Discussion, Conclusions and Future Work

In this paper different plagiarism detection tools were compared. Many of the published tools are, however, not available for use. Most analyzed tools are programmed in Java or can be started from CLI in order to integrate them into assessment systems. Moss, however, does not provide a good machine-readable result format (HTML). The often-mentioned tool Sherlock-Warwick was left out because it failed to run for assignments 1 and became unresponsive for assignment 2. This should be further investigated.

In practice a concrete threshold needs to be chosen. The threshold of 80 % was (arbitrarily) chosen for comparison in Tab. 2 and 3 because it showed to be a good value in practice. It allows (in combination with Fig. 1) to compare the different approaches to get a better feeling regarding the number of reported results and what the tools define as 80 % similar.

In several papers recall and precision or related metrics were calculated [NJK19]. However, a general problem is how to find a good ground truth – particularly for simple real world assignments. On the one hand we can never be sure to know all real plagiarisms. On the other hand the author's experience has shown that is also not easy for a human to be absolutely sure if a submission is a real plagiarism or just a very close implementation. This started to happen broadly by around 80 % similarity of all tools for assignment 1 and much earlier for assignment 2. Therefore, calculating false positives might not be possible at all. Consequently, no recall and precision are calculated but the absolute numbers of reported similar solutions as well as the number of the known plagiarisms among these are provided.

Based on the comparison for the more complex assignment 1 most tools were able to detect all known plagiarisms, however, the average reported similarity values vary a lot. For Sherlock-Sydney the average similarity per cent value on the known plagiarisms was just 22 whereas for JPlag it was 94. As most tools reported similarities from 1 to 100 % (exceptions are the ones with a hard-coded threshold) it seems to be more than just a scaling issue. These differences could also be observed for the inter-rate agreements in general where most tools do not agree well. For the very simple assignment 2 basically just consisting of a if-clause most tools reported a huge amount of highly rated similarities that is not usable in practice. Counterintuitively, this was also true for the Levenshtein approach (even without normalization). Here, maybe a hard limitation for too simple assignments does exist or the calculation formula needs to be improved. In general, it might be interesting to see if the combination of different approaches or whether other factors such as shared unique errors or considering multiple submissions at the same time yield better results. A more systematic analysis is planned (also including other languages).

Of course using plagiarism detection tools are just one way to counter plagiarism and have their drawbacks. None of these tools is a substitute for the academic's own intervention [MPH16]. Other approaches could be to (automatically) asks learners questions regarding their own solution to check their understanding [LSS21; SF20]. However, its is not clear yet, what minimum complexity of the assignments is required here.

The author would like to thank Iana Klinitska for building the basis of this research with her Bachelor's thesis.

References

[AM19] Ahadi, A.; Mathieson, L.: A Comparison of Three Popular Source code Similarity Tools for Detecting Student Plagiarism. In: Proc. Australasian Computing Education (ACE). ACM, 2019.

[CLS21] Cheers, H.; Lin, Y.; Smith, S. P.: Evaluating the robustness of source code plagiarism detection tools to pervasive plagiarism-hiding modifications./, Feb. 8, 2021, arXiv: cs.SE/2102.03997v1.

[HRV11] Hage, J.; Rademaker, P.; van Vugt, N.: Plagiarism Detection for Java: A Tool Comparison. In: Proc. CSERC '11. Pp. 33–46, 2011.

[KJH18] Keuning, H.; Jeuring, J.; Heeren, B.: A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. TOCE 19/1, 2018.

[La18] Luxton-Reilly, A.; et al.: Introductory Programming: A Systematic Literature Review. In: Proc. ITiCSE '18. ACM, pp. 55–106, 2018.

[LSS21] Lehtinen, T.; Santos, A. L.; Sorva, J.: Let's Ask Students About Their Programs, Automatically. In: ICPC. IEEE, pp. 467–475, 2021.

[Lu14] Luke, D.; Divya P.S; Sony L Johnson; Sreeprabha S; Varghese, E. B.: Software Plagiarism Detection Techniques: A Comparative Study. IJCSIT 5/4, 2014.

[Ma14] Martins, V. T.; Fonte, D.; Henriques, P. R.; da Cruz, D.: Plagiarism Detection: A Tool Survey and Comparison. In: Symposium on Languages, Applications and Technologies. Pp. 143–158, 2014.

[Mc12] McHugh, M. L.: Interrater reliability: the kappa statistic. Biochemia medica 22/3, pp. 276–282, 2012.

[MPH16] Modiba, P.; Pieterse, V.; Haskins, B.: Evaluating plagiarism detection software for introductory programming assignments. In: Proc. CSERC '16. ACM, 2016.

[NJK19] Novak, M.; Joy, M.; Kermek, D.: Source-code Similarity Detection and Detection Tools Used in Academia. TOCE 19/3, pp. 1–37, June 2019.

[SF20] Salac, J.; Franklin, D.: If They Build It, Will They Understand It? Exploring the Relationship between Student Code and Performance. In: Proc. ITiCSE '20. ACM, pp. 473–479, 2020.

[SOP11] Strickroth, S.; Olivier, H.; Pinkwart, N.: Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben? In: Proc. DeLFI '11. GI, pp. 115–126, 2011.

[SP17] Strickroth, S.; Pinkwart, N.: Eine Übersicht. In: Automatisierte Bewertung in der Programmierausbildung. Waxmann, pp. 17–38, 2017.