Quantifying Activity and Collaboration Levels in Programming Assignments

James F. Power
Department of Computer Science
Maynooth University, Ireland
james.power@mu.ie

John Waldron
School of Computer Science & Statistics
Trinity College Dublin, Ireland
john.waldron@tcd.ie

ABSTRACT

This paper presents an experience report from a third-year undergraduate compiler design course that is taught as part of a four year computer science degree. We analyse data from a study of practical assignments, evaluated in the context of take-home formative assignments and a supervised summative examination. We implement metrics to quantify the degrees of similarity between submissions for programming assignments, as well as measuring the level of activity. We present the results of our study, and discuss the utility of these metrics for our teaching practice.

CCS CONCEPTS

• **Social and professional topics** → *Computer science education*;
• **Software and its engineering** → *Compilers.*

KEYWORDS

Programming assignments, program similarity, Jaccard index

ACM Reference Format:
James F. Power and John Waldron. 2019. Quantifying Activity and Collaboration Levels in Programming Assignments. In *Innovation and Technology in Computer Science Education (ITiCSE '19), July 15–17, 2019, Aberdeen, Scotland UK*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3304221.3319769

1 INTRODUCTION

In this paper we present an experience report describing the deployment of an automated assessment and data collection system in a *Compiler Design* module given to Computer Science degree students in their third year of study. The module has a practical emphasis involving a series of six programming assignments using the *flex* and *bison* tools. With a class size of 96 during the twelve week module, over half a million lines of code were submitted. The only practical way to grade and provide timely feedback for this scale of work is automated assessment.

We present our analysis of the data extracted from electronic records of the student interaction with the learning technology used. We measure the frequency and quantity of interactions with the automated system and how this effects results. One important question with large class sizes and automated assessment is the degree of collaboration or plagiarism that may arise. We describe various different measures that can be used to compare code submissions and also algorithms and implementations to efficiently analyse approximately 12 Megabytes of source code.

In Section 2 of this paper we briefly review the background to our approach and the related work on automated assessment and similarity measures. In Section 3 we describe in more detail the compiler design module, the programming assignments and the technology we have developed to handle the large volume of submitted code. We elaborate on an algorithm to compare two different files in the context of compiler design, and its implementation. Here we also present a broad summary of the characteristics of our data. In Section 4 we reflect on our techniques and study how they relate to outcomes as measured by both formative and summative assessments. Section 5 describes dynamically generated graphical results which we believe best indicate instances of similarity between programs that merit further investigation and enable the instructor to provide relevant feedback.

ITiCSE '19, July 15–17, 2019, Aberdeen, Scotland UK

ACM ISBN 978-1-4503-6301-3/19/07…$15.00
https://doi.org/10.1145/3304221.3319769

2 BACKGROUND AND RELATED WORK

Issues related to assessment in Computer Science courses have received much attention in recent years, particularly in relation to automated assessment, and we only attempt a brief review of some of the main approaches here.

2.1 Automated assessment

A large number of different systems exist to assign marks to code, typically based on testing [6], but also potentially using quality metrics and other features [19, 21]. In addition to their use in standard introductory programming courses, automated assessment tools have recently been used for topics as diverse as databases [11], graphics [20] and functional programming [2].

A principal advantage of automated assessment is that it facilitates increased frequency of formative assessment, even for large class groups. This learning experience can be enhanced by providing more targeted automated feedback [10] and, as a side-effect, can provide a large body of data that can be used for learning analytics [8]. Even for summative assessment it has been argued that automated assessment for programming provides an environment that is closer to the code-and-test loop used in realistic programming scenarios [9].

Formative assessment typically occurs during a course and is directed at enhancing, rather than evaluating, student learning [3]. Often such assessment is awarded low marks and takes the form of take-home exams, as is the case for this paper. One of the challenges with such assessment, particularly in the context of automation,

Table 1: Summary of the programming assignments. There were six programming assignments over 11 weeks. The averages show the number of attempts per student and the average number of lines-of-code (LOC) per attempt.

	Assignment	Week	Kind	Total Students	Total Files	Avg. attempts	LOC in total	LOC per attempt
1	even	2	flex	95	857	9	17 867	21
2	comments	3	flex	91	2 640	29	63 266	24
3	plates	4	flex	91	3 848	42	178 663	46
4	roman	6	flex/bison	86	2 197	13	98 960	90
5	romcalc	9	flex/bison	84	1 911	12	161 526	170
6	calcwithvariables	11	flex/bison	71	972	7	48 081	98

is the possibility of plagiarism [4, 5] and the difficulty of ensuring academic integrity in the assessment process [15]. An alternative is to insist on supervised examinations, possibly also involving automated assessment [13].

In recent years, the use of automated assessment systems has also allowed instructors to collect a large amount of information, and some of this can be relevant in analysing student behaviour. For example, one study of students on an introductory Java course showed that the number of attempts at an exercise correlates better with performance on a final exam than the actual results of the exercise [1]. Another approach used start and end times for submissions, along with a similarity measure, as one element in identifying possible patterns of collaboration [7]. Recently, a study combined similarity measures and submission information to calculate a 'plagiarism probability' index for code [17].

One feature of many of these measures is that they are very fragile to the level of student knowledge of their use. For example, if students know the basic parameters of the algorithm being used to detect plagiarism, they can take measures to defeat it. Similarly, if students know that indices on the differences between first and final attempts are being calculated, it would be straightforward to ensure that this value is maximised.

2.2 Measuring similarity

There are many ways of measuring similarity between two documents and/or programs and the options have been extensively discussed in the literature [4, 14]. For our purposes, we were dealing with a relatively limited corpus of mixed-language submissions, and we wanted a simple and easy-to-interpret measure. We chose initially to use the Jaccard similarity index [16], later generalising this to an instance of the Tversky index.

The initial step is to represent a document as a set (of words, tokens, lines etc.). Given two programs represented by sets X and Y, their Jaccard similarity index is defined as:

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

The Jaccard index is symmetric, so $J(A, B) = J(B, A)$, which can be a problem when the sizes of the sets are not similar. For example, adding more lines to just one of the programs would decrease the similarity index for both. In order to compensate for this, we used an asymmetric index, where we normalised by the number of lines in the first program only.

That is we calculated the similarity between two sets as

$$S(X, Y) = \frac{|X \cap Y|}{|X|}$$

More formally, both of these are instances of the Tversky index [18], defined as:

$$S(X, Y) = \frac{|X \cap Y|}{|X \cap Y| + \alpha|X - Y| + \beta|Y - X|}$$

The index is defined by two parameters $\alpha, \beta \geq 0$. The Jaccard index puts $\alpha = \beta = 1$, whereas we put $\alpha = 1, \beta = 0$ so that additional (different) elements in set Y would not change the index for X.

3 CONTEXT

The module *Compiler Design* is a one-semester, 12 week, module taken by ninety-six students in the third year of a Computer Science degree at Trinity College Dublin. The module has a practical emphasis, and students study tools designed for writers of compilers and interpreters which are also useful for many other applications. The module is relevant for any application that looks for patterns in its input or has an input or command language.

The module is assessed using a summative invigilated written examination at the end of the semester worth 80%, along with six formative take-home programming assignments during the module, worth a total of 20%.

3.1 Automatic assessment system

A survey of attendees at SIGCSE 2015 showed that the majority of attendees that use automation often or extensively also use homegrown tools [19], and we use this approach as well. All the programming assignments are submitted and marked automatically using a virtual learning environment and course management system called CODEMARK which has been developed internally in Trinity College Dublin.

CODEMARK is written in a combination of PHP and HTML for the front end. The back end is a combination of the relevant programming tools, Unix utilities and shell scripts. It runs on a Citirix XenServer which is a hypervisor platform that enables the creation and management of virtualised server infrastructure.

CODEMARK allows students to submit programming assignments using PHP forms. Either code can be pasted directly into a web page form for smaller programs or uploaded in an archive file for larger tasks or those involving multiple files. It then uses the programming

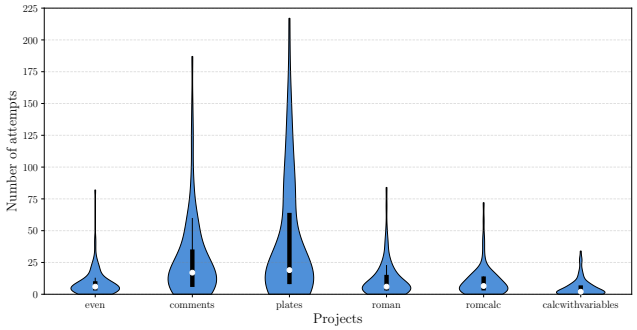

Figure 1: Activity per assignment, measured in terms of the number of attempts per student for each assignment.

tools to analyse the code for syntax errors and if none are found the resulting program is run against multiple test cases and a mark is automatically awarded based on the number of successful results.

3.2 The assignments

During the module the students study the use of the *flex* and *bison* tools [12]. Both these tools are code generators: *flex* takes as input a specification consisting of regular expressions and generates the source code for a lexical analyser, while *bison* takes as input a specification consisting of a context-free grammar and generates the source code for a parser. In both cases, the user may add snippets of C code interwoven with the specification, or whole segments of C code at the beginning or end of the file.

There are six formative programming assignments distributed throughout the module, and summarised in Table 1. There are three lexical analysis assignments (just using *flex*) and three lexical analysis/parsing assignments (using both *flex* and *bison* tools). In total the students submit six *flex* files and three *bison* files, and may make multiple submissions up to the assignment deadline.

Table 1 also list some basic metrics for each assignment to quantify the size of the data. For each assignment, Table 1 shows the number of students that attempted the assignment (a total of 96 students registered for the module) and the overall total number of files and lines of code (LOC) submitted for that assignment. Throughout this paper, we use 'code' to mean the flex/bison specifications (and embedded C code) submitted by the student, but not the code that is generated by these tools.

Table 1 also shows two averages for each assignment: the average number of attempts per student, and the average number of LOC per attempt. Overall, the CODEMARK system analysed 12,425 flex/bison source files submitted during the module, comprising 568,363 lines of code and totalling 11.8 Megabytes of data.

3.3 Processing the data

Each time a student submits an assignment for marking, the submission is logged and the files are stored by the CODEMARK system, allowing us to study the level of *activity* during the module. Figure 1 contains a violin plot for each assignment that shows the distribution of the level of activity. In Figure 1 each violin plot represents the level of activity measured in terms of the number of submissions for each student. The white dot shows the median and the black bar shows the inter-quartile range.

Figure 1 shows that the level of activity reached a peak for the third assignment, and decreased substantially thereafter. We speculate that the lower level of interactions in the first assignment was due to it being relatively easy, and the decrease after the third assignment was due to time pressures on the students as the semester proceeded. This is also reflected in the drop-off in the number of students doing each assignment from 95 to 71 as shown in Table 1.

We experimented with different algorithms to measure code similarity, essentially varying the Tversky index with different values of α and β. We did not compare submissions for different assignments and restricted ourselves to only the final submission for each assignment. Initially we developed a prototype implementation using the Unix *sort* and *comm* tools but this proved prohibitively computationally expensive, so we implemented the analyses in Python, giving a run-time in seconds. This made it feasible to perform many variations of the analysis.

In order to allow for trivial modifications to the code, we preprocessed it by removing blank lines and multiple white-space sequences, particularly in the middle of lines. We chose to split the programs based on lines (rather than words or characters) since both *flex* and *bison* lead naturally to line-oriented specifications. Because of the relative terseness of these specifications, as shown by the average LOC counts in Table 1, and the natural structuring due to using the same grammar, we felt that a word-oriented count would produce too many false positive matches. Since the variants of the Tversky index are based on comparing sets, duplicate lines in a program were automatically excluded.

Figure 2 presents a summary of the data analysed for three measures of similarity between programs: raw line counts, the Jaccard index, and the Tversky index. In each case we compare the final submission for each student for each of the 9 programs submitted, resulting in 63,814 pairwise comparisons (or 31,907 for the symmetric measures). The histograms in Figure 2 show that the Tversky index provides a greater degree of discrimination between programs for our data set.

We also analysed results with various Tversky index threshold settings and we heuristically decided on 75% to be a reasonable compromise between precision and recall. Due to the syntax of *flex* and *bison* programs we would expect a certain number of lines to be identical and not indicative of plagiarism.

As shown in the histogram in Figure 2 only a relatively small number of submissions showed a Tversky score above our threshold, suggesting that the vast majority of the students worked independently on the assignments.

4 ANALYSING THE DATA

In this section we present an analysis of the data collected during the module. This data included student marks in the assessment and final exam, student programs collected during the module, and the dates and times these files were submitted. To make the analysis manageable, we concentrate on four measures, calculated for each student and normalised as a percentage:

- **Formative**: the total marks for the *formative* assessment, comprised of the six take-home assignments described in

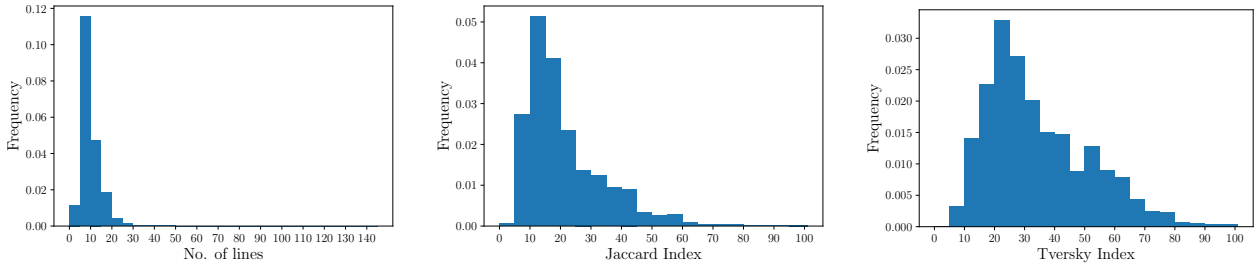

Figure 2: Distribution of similarity measures. These three histograms show the distribution of three similarity measures for the data set studied in this paper. Note that the vertical axis is on a different scale for each histogram.

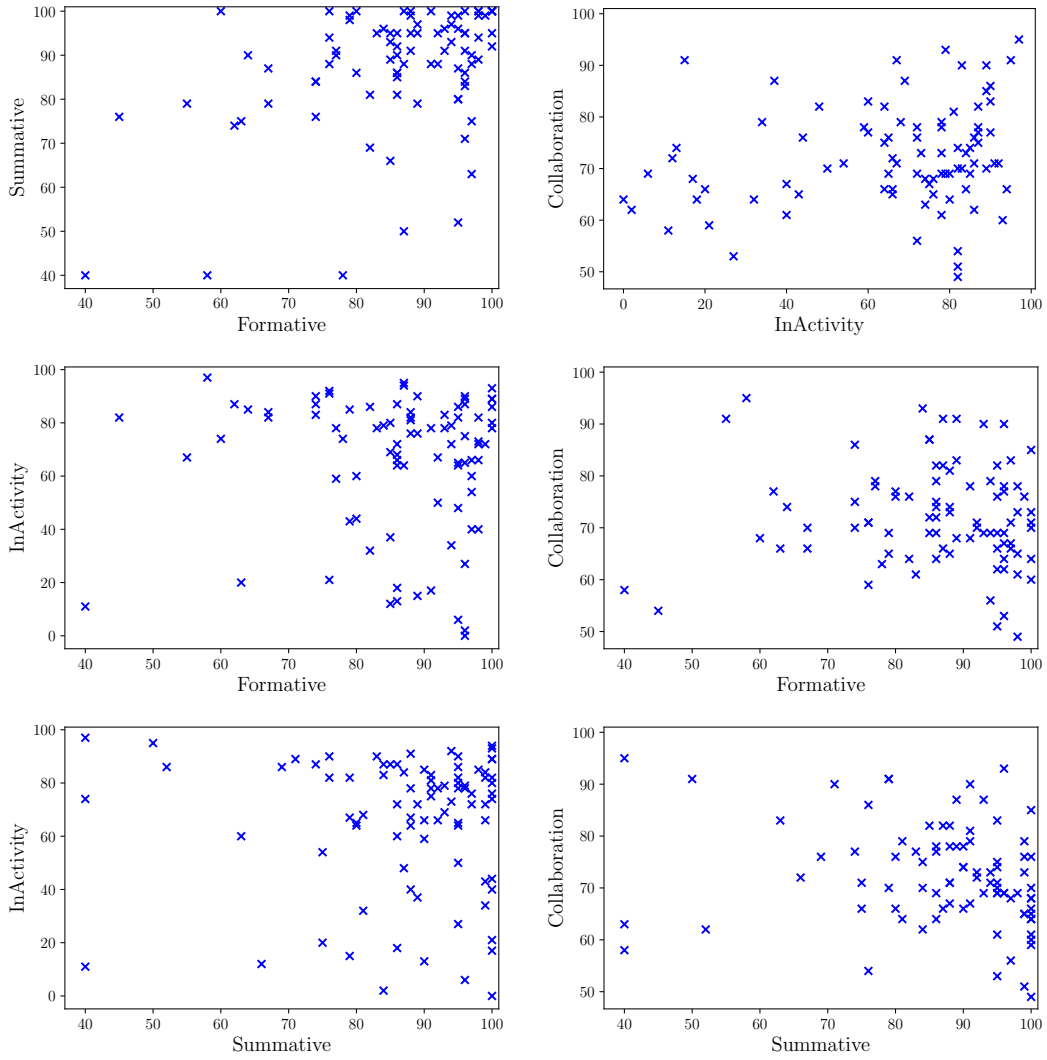

Figure 3: Six scatter-plots showing the relationships between the summative and formative marks, and the indices of inactivity and similarity. Each mark on the graph represents data for one student.

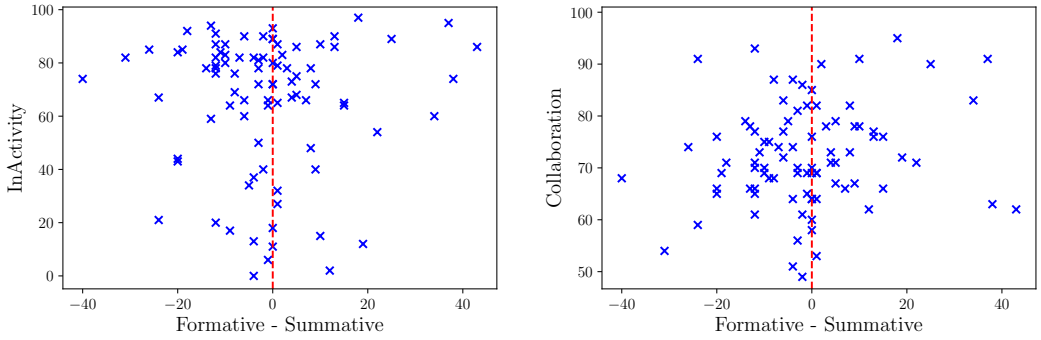

Figure 4: Two scatter-plots showing the relationship between the difference in the formative and summative marks, with the activity (left) and similarity indices (right).

Section 3. Each assignment was automatically assessed using the CODEMARK system, the mark awarded was their best mark for that assignment, and this was then averaged over the six assignments.

- **Summative**: the mark in the *summative* exam, an invigilated written examination given at the end of the module.
- **Collaboration**: a measure of the degree to which a student's code was similar that of other students. For each assignment, we took the maximum similarity measure between a student's code and that of other students, and averaged this over the number of assignments attempted by the student.
- **Activity**: We measured a student's activity by calculating the total number of submissions over the duration of the module. This was then normalised by dividing by the maximum number of submissions (407 in total for one student) and expressed as a percentage. When analysing the data we found it simpler to express this as an *in*activity measure, which is just the activity measure subtracted from 100.

Figure 3 contains six scatter-plots, showing the relationships between these four measures. Each mark on a plot represents the data for one student. Since relatively few students scored under 40% in the exam marks, we have minimised the formative and summative marks at 40% to better highlight the data. The minimum collaboration level reported for any student was 50%. By definition, the minimum activity level is 0%, the value for the most active student.

The two scatter-plots on the first row of Figure 3 depict the relationship between the exam and non-exam marks. The first plot shows the relationship between formative and summative marks. Both of these sets of marks were relatively high overall, since this is a third-year module taken by experienced students. We were particularly interested in those students who did well in the formative exams but then did poorly in the summative exam, those in the bottom-right of the first plot. We identified the 10 students nearest the bottom-right corner and examined their other measures manually, but could not distinguish any particular trend. The scatter-plot on the right of the first row of Figure 3 plots collaboration against inactivity. Again, examining those with collaboration levels over 75% did not show any identifiable trend in terms of activity.

The second and third rows of Figure 3 examine the relationships between the non-exam measures and the exam measures. We had speculated that students with high inactivity or high collaboration would exhibit poorer summative marks overall than formative marks, but a visual examination of the data in Figure 3 does not support this. While we examined individual cases, and experimented with modifications to our measures, we did not feel that more sophisticated statistical techniques were warranted on this data.

One other hypothesis we investigated was that students who exhibited a large difference between their formative and summative marks might also have distinctive patterns in their levels of activity or collaboration. To visualise this, Figure 4 has two scatter-plots, showing the relationship between the non-exam measures and the difference in the two exam measures. Students on the right of these two plots are those who increased their mark between formative and summative exams. Again, there is not a clear relationship between these students and either the inactivity or collaboration measures. We performed a one-sided Mann-Whitney U test using the difference ($<= 0$ and > 0) as a classifier for both activity and collaboration, and confirmed that this is not an informative classifier in this case.

5 COLLABORATION GRAPHS

The overall measures summarised in Section 4 did not allow us to make general statements regarding the predictive capabilities of the measures we used. However, the results are still interesting on an individual level, and could be used to provide information to the instructor during the course.

The CODEMARK system provides a large amount of data, and it can be difficult to pick out interesting features on a per-student basis. After experimenting with a number of possibilities, we have found it most useful to visualise the levels of collaboration using a *collaboration graph*, an example of which is shown in Figure 5. This graph is generated automatically from the data analysis elements of our code, and the visualisation is produced using the *Graphviz* graph visualisation software using the *circo* layout engine.[1]

[1]Graphviz version 2.38.0, available from https://graphviz.org/

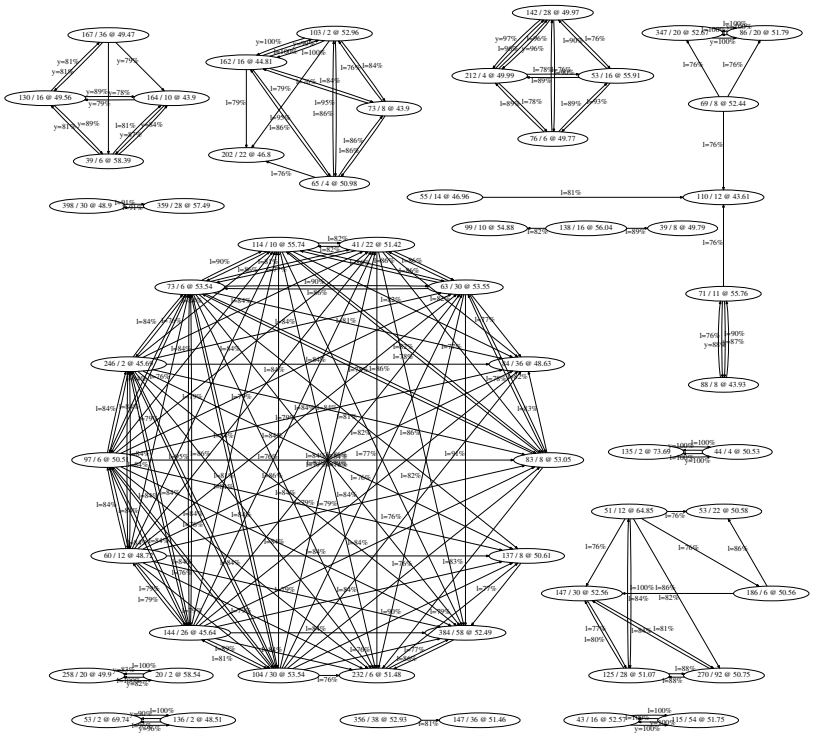

Figure 5: An example of a collaboration graph for the fifth assignment. Each node represents a student, and each (directed) edge represents the maximum similarity level from one student to the other, thresholded at 75%.

Figure 5 contains a directed graph, where the nodes represent students, and the edges represent a collaboration from one student to another, based on our collaboration measure. We have deleted identifying details from this example, but the student number would normally be included in each node. The collaboration has been thresholded at 75%, as discussed earlier in Section 3.3, but this can be changed, with a lower threshold producing more edges.

As can be seen from the overall structure of Figure 5, the collaboration graph quickly identifies groups of students who have high levels of similarity between their submissions, and the weighting on the edges gives an indication as to the level of collaboration. The graph in Figure 5 shows one large collaboration on the bottom-left, a heavily connected network of 14 students. A number of other smaller collaborative groups are also shown, as are collaborations between two students.

When analysing the collaboration networks we found it useful to have extra information available to help with identifying more likely instances of copying. Thus, in each node we print activity information, showing the overall level of activity, the level of activity for this assignment, and the time (in days since the start of the module) of the last submission. For example, the node in the bottom-left of the graph is labelled "356 / 38 @ 52.93", meaning that this student has 356 submissions overall, 38 in this project, and the last submission was toward the end of day 52 of the module.

In general we interpret high activity and early submission as more favourable indicators of a student's engagement, and are interested in situations where these metrics differ between collaborating students. We would not advocate using these metrics on their own, but rather as an indicator of situations that merit further investigation.

6 REFLECTION

In this paper we have described our use of an automated system, CODEMARK, to provide assessment and data analysis for a third-year *Compiler Design* module. As such, it contributes to a growing body of knowledge on the use of such systems, both in introductory programming courses and at more advanced levels [2, 11, 20].

It should be noted that these results are based on a single group of students, so generalising these results would require further studies using different groups. We hope to explore this issue in future work to provide a broader context for our analysis.

Our work demonstrates the feasibility of deploying such software for a large class group with an intensive assessment schedule. It also indicates the difficulty of choosing suitable metrics that can provide insight on the large body of data collected. Because of our analytical techniques and associated software implementation, it is now practical for the instructor to see this information in real time and to conduct appropriate interventions. It is our thesis that the best policy must be based on a well-informed personal intervention after manual inspection of the data, and our work here is intended to provide a foundation for that approach.

REFERENCES

[1] Alireza Ahadi, Raymond Lister, and Arto Vihavainen. 2016. On the Number of Attempts Students Made on Some Online Programming Exercises During Semester and their Subsequent Performance on Final Exam Questions. In *ACM Conference on Innovation and Technology in Computer Science Education*. Arequipa, Peru, 218–223.

[2] Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. 2018. The Effect of a Web-based Coding Tool with Automatic Feedback on Students' Performance and Perceptions. In *49th ACM Technical Symposium on Computer Science Education*. Baltimore, MD, USA, 2–7.

[3] Benjamin S. Bloom, Thomas Hasting, and George Madaus. 1971. *Handbook of formative and summative evaluation of student learning*. McGraw-Hill, New York, USA.

[4] Paul Clough. 2003. *Old and new challenges in automatic plagiarism detection*. Technical Report. Department of Information Studies, University of Sheffield.

[5] Charlie Daly and Jane Horgan. 2005. A Technique for Detecting Plagiarism in Computer Code. *Comput. J.* 48, 6 (2005), 662–666.

[6] Charlie Daly and John Waldron. 2004. Assessing the Assessment of Programming Ability. *SIGCSE Bull.* 36, 1 (March 2004), 210–213.

[7] Arto Hellas, Juho Leinonen, and Petri Ihantola. 2017. Plagiarism in Take-home Exams: Help-seeking, Collaboration, and Systematic Cheating. In *ACM Conference on Innovation and Technology in Computer Science Education*. Bologna, Italy, 238–243.

[8] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *ITiCSE on Working Group Reports*. Vilnius, Lithuania, 41–63.

[9] An Ju, Ben Mehne, Andrew Halle, and Armando Fox. 2018. In-class coding-based summative assessments: tools, challenges, and experience. In *23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. Larnaca, Cyprus, 75–80.

[10] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *ACM Conference on Innovation and Technology in Computer Science Education*. Arequipa, Peru, 41–46.

[11] Anthony Kleerekoper and Andrew Schofield. 2018. SQL tester: an online SQL assessment tool and its impact. In *23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. Larnaca, Cyprus, 87–92.

[12] John Levine. 2009. *Flex & Bison*. O'Reilly Media, Sebastopol, CA, USA.

[13] Phil Maguire, Rebecca Maguire, and Robert Kelly. 2017. Using automatic machine assessment to teach computer programming. *Computer Science Education* 27, 3-4 (2017), 197–214.

[14] Chanchal Kumar Roy and James R. Cordy. 2007. *A Survey on Software Clone Detection Research*. Technical Report No. 2007-541. School of Computing, Queen's University at Kingston, , Ontario, Canada.

[15] Judy Sheard, Simon, Matthew Butler, Katrina Falkner, Michael Morgan, and Amali Weerasinghe. 2017. Strategies for Maintaining Academic Integrity in First-Year Computing Courses. In *ACM Conference on Innovation and Technology in Computer Science Education*. Bologna, Italy, 244–249.

[16] Kwangho Song, Jihong Min, Gayoung Lee, Sang Chul Shin, and Yoo-Sung Kim. 2015. An Improvement of Plagiarized Area Detection System Using Jaccard Correlation Coefficient Distance Algorithm. *Computer Science and Information Technology* 3, 3 (2015), 76–80.

[17] Narjes Tahaei and David C. Noelle. 2018. Automated Plagiarism Detection for Computer Programming Exercises Based on Patterns of Resubmission. In *ACM Conference on International Computing Education Research*. Espoo, Finland, 178–186.

[18] A. Tversky. 1977. Features of similarity. *Psychological Review* 84, 14 (1977), 327–352.

[19] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In *47th ACM Technical Symposium on Computing Science Education*. Memphis, Tennessee, USA, 437–442.

[20] Burkhard C. Wünsche, Zhen Chen, Lindsay Alexander Shaw, Thomas Suselo, Kai-Cheung Leung, Davis Dimalen, Wannes van der Mark, Andrew Luxton-Reilly, and Richard Lobb. 2018. Automatic assessment of OpenGL computer graphics assignments. In *23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. Larnaca, Cyprus, 81–86.

[21] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Judith Bishop. 2015. Educational Software Engineering: Where Software Engineering, Education, and Gaming Meet. In *Computer Games and Software Engineering*. CRC Press, 113–133.