Managing Plagiarism in Programming Assignments with Blended Assessment and Randomisation

Steven Bradley
School of Engineering and Computing Sciences
Durham University
UK
s.p.bradley@durham.ac.uk

ABSTRACT

Plagiarism is a common concern for coursework in many situations, particularly where electronic solutions can be provided e.g. computer programs, and leads to unreliability of assessment. Written exams are often used to try to deal with this, and to increase reliability, but at the expense of validity. One solution, outlined in this paper, is to randomise the work that is set for students so that it is very unlikely that any two students will be working on exactly the same problem set. This also helps to address the issue of students trying to outsource their work by paying external people to complete their assignments for them. We examine the effectiveness of this approach and others (including blended assessment) by analysing the spread of similarity scores across four different introductory programming assignments to find the *natural similarity* i.e. the level of similarity that could reasonably occur without plagiarism. The results of the study indicate that divergent assessment (having more than one possible solution) as opposed to convergent assessment (only one solution) is the dominant factor in natural similarity. A key area for further work is to apply the analysis to a larger sample of programming assignments to better understand the impact of different features of the assignment design on natural similarity and hence the detection of plagiarism.

Keywords

programming, assessment, plagiarism, automated grading, blended learning

1. INTRODUCTION

1.1 Plagiarism

Plagiarism has long been recognised as a problem in computer science education, and for programming in particular [16, 15]. In one study [8], 33% of students said they had plagiarised and 30% said a computer program they had written had been a source for plagiarism. Only 64% of staff in the same study had caught a student trying to plagiarize and 12% of the staff had never heard of a case of plagiarism, suggesting that a lot of plagiarism goes unnoticed. At the heart of the problem are the issues of reliability and validity of assessment [19]. For many academics programming coursework feels like a more valid way of assessing students than a written exam, but according to one of the interviewees in a study [22]

> "We do exams because of plagiarism, there is no other reason for doing an exam"

If students can cheat in an assessment then the assessment is unreliable, ergo coursework is unreliable. Much effort has therefore been spent on plagiarism detection tools for programming such as MOSS [5] and JPlag [21]. A comparison of plagiarism tools in 2010 [13] identified 18 separate projects/tools, and since then more work has been carried out on new approaches to identifying plagiarism [9, 11].

1.2 Plagiarism Detection and Natural Similarity

Typically a plagiarism detection tool will give a ranked list of submission pairings ordered in terms of their similarity with an adjustable cutoff point for inclusion in the list. There is, however, no universal of similarity beyond which we can say "this work has been plagiarised" as it may be that two programs could have been produced independently and yet have substantial similarities. In practice, our approach to using a plagiarism detection tool is to look through the ranking and decide from an expert academic standpoint whether there is evidence for plagiarism beyond reasonable doubt. The ranked list of pairings is traversed from high similarity to low until the similarities encountered could reasonably have occurred without plagiarism. This defines a threshold which we define to be the *natural similarity* for the assignment and is potentially affected by many factors, including the following:

- if students are taught or required to follow particular coding standards the natural similarity will be higher;
- if students are required to implement methods/functions with particular names and calling conventions (e.g. implementing a Java interface) the natural similarity will be higher;
- if students have access to code examples very similar to those required for the assignment — either directly

Koli Calling November 24-27, 2016 Koli, FInland

ISBN ACM 978-1-4503-4770-9/16/11…$15.00

DOI: http://dx.doi.org/10.1145/2999541.2999550

from the teacher, or from elsewhere (possibly if the assignment is similar to standard exercises) — the natural similarity will be higher;

- moving along the continuum between *convergent* assessment (where there is only one correct answer) and *divergent* assessment (no one correct answer, but rather an overall measure of quality) [19] will reduce the natural similarity;

- the more insensitive the detection tool is to "standard" obfuscation techniques (e.g. renaming of variables, adding comments) the higher the natural similarity there will be. This may possibly counteract other dependencies, e.g. coding standards and required method names.

While natural similarity can be defined numerically for a given assignment, it is still partly subjective, based on the opinions of the assessor and the level of proof required to accuse a student of plagiarism.

1.3 Automated Assessment

Aside from plagiarism, there is another potential problem with coursework as opposed to written exams: the amount of time spent marking. The development of systems for automated assessment of programming assignments has taken place in parallel with the progress in plagiarism-checking tools, with 17 different systems reviewed in 2010 [14]. While the most commonly stated reason for adopting automated assessment is the ability to give almost immediate feedback to students, assisting them in their learning, the reduction in the time spent by the tutor can also be a key driver. Reliability of assessment should also increase, because of the objective application of assessment criteria, but this is less often referred to in the literature.

In order to meet the needs of the assessment system, the task needs to be very clearly specified, which some have argued as constraining to the nature of the assessment, possibly reducing its validity. The approach that we have used for automated assessment involves getting students to implement Java interfaces, with the submitted code then exercised via JUnit. As discussed above, it would be expected that this would increase the level of natural similarity between submissions.

Why do we care about natural similarity? A high level of natural similarity within an assignment makes it hard to detect plagiarism because it is more likely that similarities could have occurred by chance. Plagiarised work is still identified, but potentially so are pairs of submissions that are not plagiarised, and which are reported as false positives. Later in the paper (section 3) we examine how different types of assessment affect the natural similarity in practice, through a study of real programming assignments. First, in section 2 we look in more detail at the use of coursework and exams plus manual and automatic assessment and examine the notion of blended assessment, describing an approach we have developed which is aimed at increasing reliability of assessment whilst maintaining or increasing validity. This approach is evaluated alongside others in the study (section 3), and conclusions are drawn in section 4, along with suggestions for further work.

2. PROPOSAL: BLENDED ASSESSMENT AND RANDOMISATION

We propose two approaches to programming assignments: blended assessment to improve validity (section 2.1) and randomisation to improve reliability (section 2.2).

2.1 Blended Assessment

To strike a balance between the reliability of formal written exams and the validity of coursework exercises, the use of laboratory exams has been discussed [10], where students carry out programming exercises, rather than written exams, in more constrained exam-like conditions. We have used this approach in the past, but concerns remain about the validity of the assessment approach (as well as its reliability, which is discussed in section 2.2). What can a novice programmer be expected to achieve in 75-180 minutes, the range of lengths of exam times reported in the study [10]? Some academics also expressed concerns that if a student gets stuck early on in the exam session then they are unable to make progress at all, so end up with very few marks (and a lot of stress). This situation does not reflect how they have learnt (or how they would practice in the real world), where other students or tutors can help point out problems. We have probably all been in a situation where we could not identify an obvious but simple error that was staring us in the face. To address some of these issues, we have used an approach that could be referred to as blended assessment. The term *blended assessment*, although sounding somewhat fashionable, has potentially multiple meanings — and two of the meanings apply here.

The idea of blended learning has been around for quite a while and, although the definition is not universally agreed, most definitions include the idea of combining instructional approaches, typically including face-to-face and computer mediated/on-line.[4, 6]. Blended assessment, on the other hand, is a newer term which has a wider variety of interpretations. One paper [18] describes an environment in which traditional-style written exams are blended with e-assessment techniques for managing grading the papers, reporting to students and analysis of results. In another study [20] on-line multiple-choice tests are blended with follow-on written responses, while another looks at different ways of integrating of on-line exams within courses [1].

2.1.1 Blending Coursework and Laboratory Exam

Our first aspect of blending is to combine in one assignment the notion of a standard piece of 'take home' coursework with a laboratory exam. Cutts et al. [10] identified the use of published scenarios, or even the whole exam paper, in advance of the time-constrained exam. We published an initial problem specification, with a Java interface to be implemented, in advance of the test, explaining to the students that they would be required to adapt their code to new requirements, to a new interface, in their scheduled time slot. The motivation here was threefold:

1. To broaden the scope of what could be expected of students

2. To deter students from plagiarising the 'take home' portion, as they would have to understand the code to be able to adapt it

3. To reduce the likelihood of students getting completely stuck in the time-constrained exam

Marking of the work is automated relatively easily using JUnit via the prescribed interfaces.

2.1.2 Blending Automatic and Manual Assessment

Automated assessment is often used to give quick feedback to students without intervention from the tutor, typically through a web-based system e.g. Web-CAT [12]. This has some clear pedagogical advantages, but also some problems if the students rely on resubmission too much. [17]. Our approach has been to get lab demonstrators to provide face-to-face feedback in weekly lab sessions, but to use automatic assessment for the larger summative exercises.

Students can be uncomfortable with automated assessment because of its binary nature: if the code doesn't run exactly as required then no marks can be awarded. By testing different parts of the functionality separately it is perfectly possible to get a finer granularity, but sometimes the student's work essentially fails to get off the ground, for instance if it doesn't compile. This sounds like a a situation where it would be perfectly reasonable for the student to be awarded zero marks, but what if the student's work compiles by itself, but not within the testing environment. Typical causes for this that we have seen (within the context of Java) are,

1. Although all of the functionality required by the interface is in place, the class does not include the "implements" declaration

2. The student has used the wrong version of the interface

3. The compiler used by the student is not compatible with the marker (too old or too new)

4. The specified constructors have not been provided: Java does not allow constructors to be specified within an interface. The main alternative to constructors — factory methods — need to be static, and hence cannot be specified in Java interfaces prior to Java 8.

There are also lots of ways that the code might compile but fail to work for a very simple reason, for instance if a "set" method used by the tester does not work properly.

Some of these issues can be dealt with programmatically, and we have used Java reflection to identify when constructors are not present and to add them in by generating a new version of the source code with the correct constructor included (usually just a nullary constructor). In this way partial credit can be given automatically for programs that are very nearly correct. Sometimes though, more work is needed to make the class compile with the tester (or even to compile at all) which really can only be carried out by human intervention. The two main challenges in implementing this approach is to integrate the feedback provided by the automated assessment tool with the human-produced feedback, and how to aggregate the marks between the two. The solution that we have used is via a web-based statement bank tool that was developed previously for expediting the process of manual assessment. Statement bank systems are fairly common, but the main distinguishing features of our system are that

- new statements can be added to the bank of statements 'on the fly' either by adapting existing statements, or through a web API,
- marks can be associated automatically within statements by regular expression matching: including an integer in brackets within the statement will interpret this as a mark to be included in the total and
- the marking scheme structure of an assignment can be represented, so that
  - statements previously made within that section can be prioritised for inclusion and
  - maximum marks for a section can be checked against marks awarded.

An initial pre-marking process is run which identifies whether the submitted code compiles, and tries to apply programmatic fixes. Marks are awarded simply for the code compiling as required, and none if not – this avoids the contentious issue of negative marking, and allows the human marker to award partial scores if appropriate, reflecting the changes that have had to be made. We claim that this increases the validity of the test with a relatively small impact on reliability. Manually adding the relevant comments (with marks) from the statement bank, and seeing the marks associated with similar comments, certainly adds confidence to the feeling that the human intervention in the marking is consistent.

Comments are added from the automated assessment system through naming of the test methods that are applied, for example in this test

```
@Test
public void isValid_returns_true_if_valid_2(){
  assertTrue(met.isValid());
}
```

Here the test is applied to the object `met` which has been constructed to be valid, and depending on the result of the test one of the following two statements are added to the student's feedback within the relevant section of the marks scheme.

```
isValid returns true if valid [max 2]  YES (2)
isValid returns true if valid [max 2]  NO
```

In this case the mark scheme was sectioned according to the different classes that the students were required to implement, so it is clear from the context which `isValid` method is being referred to.

As well as simple tests like this, structural properties of the code can be tested through reflection and assessed automatically. For instance, one fairly common mistake that students make is to declare variables as fields which should really be local variables within methods, and this can be tested for with assertions about the number and types of fields within a class. Because automatically generated statements can be included alongside statements written/assigned by humans, it is easy to have parts of the assessment done by hand e.g. quality of user interface, and because this is all done through the web different parts of the assignment can be completed by different human and automatic markers, allowing for blending of a wide variety of assessment methods within the assignment.

2.2 Randomisation

In laboratory exams, as well as the usual the usual factors associated with exams such as the length of the exam, whether is it seen/unseen, open/closed book, one of the main issues identified by Cutts et al. [10] was to deal with classes where it is infeasible to have all of the students sit the laboratory exam at the same time, due to timetabling or resource constraints (availability of labs). Having students sit the same laboratory exam at different times risks affecting reliability through possible plagiarism, whilst having the students sit different exams risks reliability through testing different things. Different solutions were adopted, but the most usual was to set slightly different versions of the exam to different students, sometimes based around a common scenario. In some approaches each sitting had its own version, but another approach was to have different versions randomly scattered across the students, to avoid the risk of copying 'over the shoulder' within the exam itself.

Our approach, which is novel as far as we know, is to have a more fine-grained randomisation of the assignment by having a range of choice points within the work (e.g. a choice of methods to implement) and have each choice made randomly. With only a few choice points the combinations mount up, so that it is unlikely that any two students have the same piece of work to complete. In the context of plagiarism, this has two advantages:

- for a student to copy a full solution they would need access to a wide range of other solutions, which would need to be adapted to work together;
- if an anonymous contract cheating request is found it is possible to identify nearly uniquely which student has made the request.

Clearly there are reliability issues with giving different students different pieces of work to complete, but that is traded off against the expected gain in reliability due to reduction in the possibility of plagiarism. Another difficulty with this approach is in automatic assessment: with randomly generated problems, what test cases should be applied? Our solution to this is to seed the psuedo-random number generator with the hash code of the student username, concatenated with a string describing the choice being made. The student downloads a generated Java interface which specifies their particular version of the task. Test cases are then prepared for each of the adaptations, and are selected at assessment time using the same seeded pseudo-random sequences. This idea is used in procedural content generation (PCG) in games [23], notably in the influential and innovative 1984 game 'Elite' [2, p113]. Because the individualised work has to be downloaded, this can be time-restricted and password-protected to ensure that downloads can only take place at the right time and place i.e. in the laboratory exam setting.

A very simple example is as follows, for a class to represent a position on a map specified through and interface `PositionInterface`. We released an initial specification via a Java interface, to be implemented in the students' own time, which required get and set methods, a `toString()` method and a method

```
boolean northOf(PositionInterface p);
```

The revised (personalised) version of the interface, which was to be implemented by the students in the laboratory exam setting, was a psuedo-randomly assigned choice between two methods `eastOf` and `westOf`. The detail of the requirement was included with the javadoc comments within the interface e.g.

```
/** Compare the longitude with another position
 *  @param p The position to be compared with
 *  @return true if and only if this object is
 * east of (i.e. has higher longitude than) the
 * parameter object
 */
boolean eastOf(PositionInterface p);
```

More complex examples were used, including getting students to validate different characteristics of an input, or finding different statistical summaries (e.g. average/ minimum/ maximum) of different properties (e.g. hours of sunshine/ days of frost/ mm of precipitation) selected by different characteristics (e.g. in frost-free months). A balance has to be struck between improving reliability by reducing the possibility of plagiarism, versus reducing reliability by assessing the students on different problems.

Figure 1 shows how the processing of setting and marking a student assignment is carried out.

3. STUDY

3.1 Definition of Research Question

We have adopted blended assessment and randomisation as described above primarily as a means to increase reliability, firstly by reducing the inclination to plagiarise because

- building on plagiarised take-home work during the laboratory exam would be difficult without good understanding and
- carrying out contract-based plagiarism on personalised specification increases the likelihood of discovery.

However, analysing students' inclination to plagiarise on a particular piece of work is challenging as there are clearly serious risks to the students in answering accurately. Whilst anonymisation is possible and has been used elsewhere [8], dealing with a particular assignment rather than a general propensity to plagiarise raises further issues. This is outside the scope of the paper, but is suggested as an area for further work.

A more easily measurable outcome, which is still very pertinent, is the level of natural similarity (see section 1). The higher the natural similarity in an assignment the more plagiarism (in terms of similarity measure) can be carried out without detection, or at least accusation, because of the likelihood of false positives. If we assume that a relatively small proportion of the student population plagiarises for a particular assignment, which is not unreasonable if the reported 33% of students that have ever plagiarised [8] applies, then the distribution of the similarities across all submissions gives a good indication of what the natural similarity would be, so we examine this as a proxy for natural similarity.

3.1.1 Research Question

What is the variation in the distribution of similarities across different introductory programming assignments, based on the following parameters:

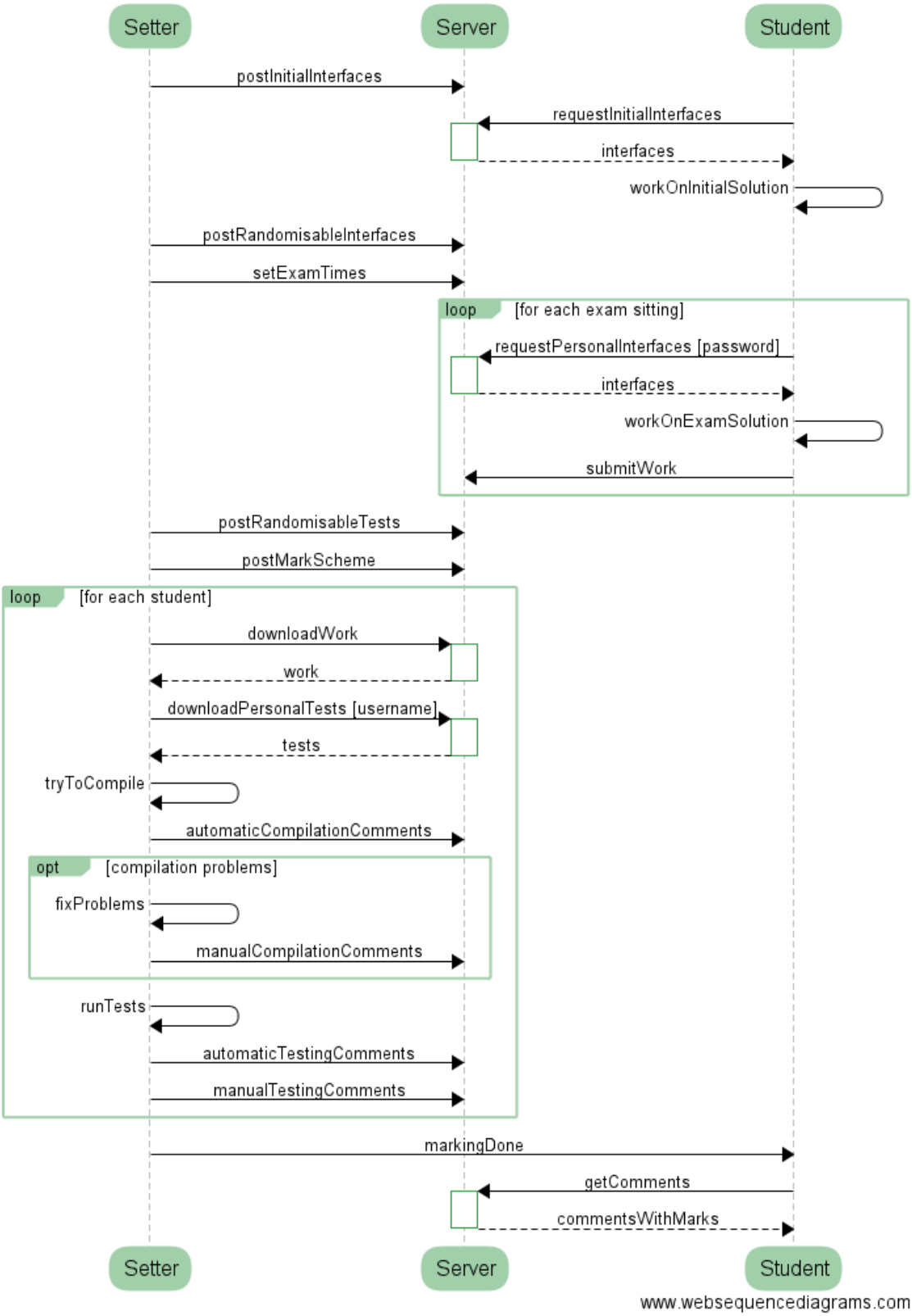


Figure 1: Sequence diagram for setting and marking process with blending and randomisation

1. the size of the group;

2. whether the assignment is constrained (i.e. a laboratory exam), unconstrained (i.e. take-home work) or blended (see section 2.1.1);

3. whether the assignment content was randomised per student, per session, or not at all;

4. whether the assignment is convergent or divergent [19];

5. whether the structure of the code is specified precisely (e.g. via a Java interface)?

3.2 Background

Two different introductory programming modules were investigated: one was for first year undergraduates (UG) from a range of disciplines, including computer science; the other was for a group of conversion MSc postgraduate students (PG) i.e. those with a degree, but not from a computer science background. Both modules were taught in Java by the same lecturer, using similar materials but over different timescales. The UG module was taught over a whole academic year alongside other modules, whilst the PG module was taught intensively over five weeks. Both modules were continuously assessed with no exams, although a written closed-book test formed part of each module. Students submitted work electronically and no resubmission was allowed. Both modules had a related set of exercises to complete which were largely formative, although the UG class had a small summative weighting on these exercises to encourage them to attend weekly practicals. The modules were taught in Java as an introduction to object-oriented programming using an objects-first approach [3]. Automated assessment was used for all of the UG assignments but not for the PG assignment, and only to return the summative marks, not for formative feedback. Two years' worth of data were collected for the UG module, but just one year of PG data. The UG assignments are labelled here by the week in which they were taken (e.g. UGw22) but this does not imply a chronological ordering because the assignments were taken across academic years.

3.3 Method

Programming assignment submissions for the modules were each compared with JPlag [21] using the `-vl` flag to save all of the comparison data. This data was then fed into R to plot histograms for the percentage similarity scores obtained. The assignments were categorised according to the parameters specified in the research question as follows.

3.3.1 PG

This postgraduate assignment, completed by 22 students, was based on historical weather station observations from the UK MetOffice. It was entirely a take-home exercise, so was **unconstrained** and was **not randomised**. A fair amount of the assignment was convergent, asking students to find average temperatures, wettest months etc but also included a requirement to develop a loosely specified text-based user interface which was more divergent, so the assignment was overall classified as **partly divergent**. Because of the relatively small group size the investment in automated assessment was not felt worthwhile and **no Java interfaces** were specified, although an informal list of the required functionality was given. This was the final summative assessment for the module.

Table 1: Variation in mean and submission similarity distribution between sessions within UGw18

Session	Mean	Standard Deviation
1	35.4	11.4
2	42.9	10.5
3	41.9	10.4
4	42.1	9.3
5	42.5	9.8

3.3.2 UGw18

Written as a laboratory exam, this **time-constrained** two hour test was open-book and networked, but the 140 students were not allowed to communicate with each other. Because the test ran inside one of the regular practical timetable slots, five sessions were run in one week. The same scenario, a hire shop, was used for all of the the sessions, but different questions included for each session, so that it was essentially **randomised per session**. Assessment was largely **convergent** and automated, with **Java interfaces specified**.

3.3.3 UGw22

This was the final UG summative assignment for the year, using the game of Oware as its basis. All 140 students had to develop a board, implement the rules and write an 'intelligent' computer player plus text-based user interface in whatever way they wanted. As such, the assessment was **largely divergent**, and **unconstrained** with **no randomisation**. Automated assessment was used and **Java interfaces were specified** which allowed different computer players to play against each other, which was used during automated marking. The quality of the user interface was assessed manually, so that the feedback was blended between automated and manual.

3.3.4 UGw12

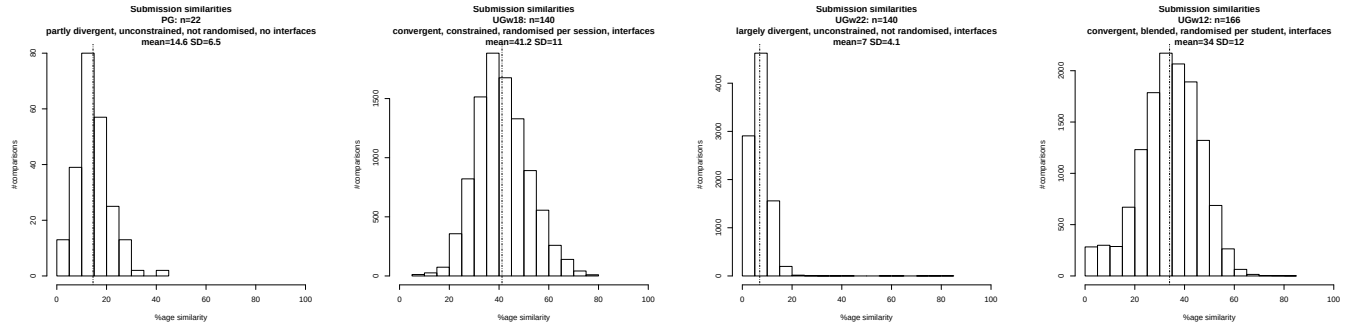
166 UG students took this assignment, which was based around the historical weather station data also used in the PG assignment, but was more limited in scope and fairly **convergent**. The assessment was **blended**, with **interfaces specified** in advance of a laboratory exam, in which new interfaces were **randomly generated for each student** in line with the process outlined in section 2.2 .

3.4 Results

Histograms of the similarities for the different assignments are given in figures 2.

All of the distributions individually pass the Shapiro-Wilk normality test, and the mean and standard deviations as shown on the figures are a reasonable summary of the data. Looking in more detail at assignment UGw18, which was replicated with variation over five different sessions, it is interesting to note the means and standard deviations as the sessions progress, see table 1

It should be noted that, due to the similarity of the problem set for PG and UGw12, there is the possibility of some similarity due to plagiarism between the two groups. However, the groups were very distinct and had little to do with each other, and the two pieces of work were completed within a short time of each other.



(a) PG (b) UGw18 (c) UGw22 (d) UGw12

Figure 2: Similarity histograms for different programming assignments, with mean similarity

3.5 Discussion

Each of the parameters of the research question are addressed in turn. While it would be possible to draw scatter plots and carry out regression analyses, given the small sample size (four assignments) that would seem to be expecting too much of the data.

3.5.1 Group Size

There is no evidence that the size of the group affects the distribution of similarities. The smallest group size by far (PG) does not have the largest or smallest average and the other group sizes are broadly similar but with markedly different similarity averages.

3.5.2 Constrained vs Unconstrained vs Blended

The two unconstrained assignments have the lowest similarity averages, and the constrained exercise has the highest similarity average, with the blended assignment in the middle. This seems to suggest that the more constrained the assessment is, the more similar the submissions. This is counterintuitive: we would expect a higher similarity from an unconstrained exercise because the opportunity to copy should be much higher in a take-home assignment, or one which includes a take-home component. From this we deduce that the effect on the average similarity of the level of constraint of the assessment is small compared with other factors.

It is worth noting, however, that the constrained laboratory exam was based on five separate sessions, with a shared scenario but different questions in each session. It could be argued that this is not really as constrained as we might hope, as there is a possibility that students in later sessions could benefit from knowing the scenario and/or type of questions that were asked. Table 1 does show a significant difference between session one and the other sessions, which took place later in the same week. This can either be put down to variation in the students within the group, variation within the questions asked, or the fact that it was the first session. It is very tempting to suggest that the later students knew what was going to happen, so could prepare more, so had higher average similarities, reducing the desired impact of having a laboratory exam rather than a take-home exercise, particularly given how similar the other sessions were to each other. The difference of the first session is statistically significant if we treat the similarities as separate observed events, which is questionable. However, the second session took place immediately after the first session, which seems to put this conclusion into question as there would have been so little time for students to benefit from knowing the contents of the first test. One thing can be said though, even the first session had a much higher average similarity (35.4) than the unconstrained assignments (14.6 and 7.0), so we can still reasonably deduce that the level of constraint is not a significant factor because this runs counter to the effect we might expect.

3.5.3 Randomisation

In our study randomisation only occurred within a laboratory exam setting which, as we have already seen, seems to have high average similarities (whether because of the constraint or some other factor). The per-student randomisation assignment UGw12 has an mean similarity of 34.0 which is lower than the overall average of the the per-session randomisation of UGw18. UGw12 is remarkably similar in distribution to session one of UGw18 (mean = 35.4) which might point towards per-student randomisation overcoming the effect of 'leakage' between sessions, but the two assignments are different enough for this to be explained away as coincidence.

3.5.4 Convergent vs Divergent

This parameter seems to have the strongest effect on the average similarity. The largely divergent UGw22 (mean = 7.0) has lower average similarity than the partly divergent PG (mean = 14.6) which in turn has lower average similarity than the convergent UGw12 (mean = 34.0) and UGw12 (mean = 41.2). This is in line with intuition, in that the more similar the solutions are expected to be (convergent) the more similar they display. The only surprising thing about this result is the extent to which it dominates other potential effects.

3.5.5 Interfaces

We would expect assignments that were tightly specified via interfaces to have higher similarity, but this is not borne out by the data. UGw22 has the lowest mean similarity, despite having specified interfaces. Again, it may be that there is an effect that is dominated by convergence vs divergence, so cannot be distinguished through such a small sample size. This may also be a facet of the plagiarism detection tool in use, which is designed to be insensitive to renaming of meth-

ods and variables. Specifying, via an interface, exactly what the methods should be called should have no effect, leaving only the similarity due to parameter and return types liable to impact on reported similarity.

4. CONCLUSIONS

All of these conclusions need to be tempered with the fact that the sample size of assignments is small, so further investigation is required. However, there are enough useful indicators to warrant such further investigation and the measures of similarity that we have been using do seem to point to real differences between assignments. Looking at the similarity distributions of the assignments studied shows large variation, not always in ways that are expected. We think this shows that is is a useful measure, although it is based on assumptions that the total amount of plagiarism is relatively small, so does not skew the distribution too much, and that the similarity measure produced by the plagiarism tool is a good indicator of plagiarism. Fundamentally, the more similarity shown by non-plagiarised solutions, the more plagiarism might go unnoticed and unprosecuted. Whilst the possibility of plagiarism tends to reduce the reliability of assessment, this is often held in tension with the need for valid assessment, which many believe cannot be achieved only through time-constrained written exams.

4.1 Variations in Natural Similarity of Assignments

We defined the *natural similarity* of an assignment to be the threshold for submission similarity score for non-plagiarised assignments. This is not to say that plagiarised assignments would necessarily have a higher similarity score than this, or that all scores higher than this are plagiarised, but that any similarity score under the similarity threshold could reasonably have been achieved without plagiarism. Assuming that the observed distribution of the the cohort is a good basis for the natural similarity Key indications from our analysis of similarity distrubutions are

- Divergent/convergent assessment [19] is the most important factor in the similarity distributions of the assignments, with divergent assessment leading to lower levels of similarity. It could be argued that divergent assessments are also more valid as they can examine higher level skills, so overall this indicates strongly that divergent assessment is better with respect to reliability and validity.

- Constraining assignments i.e. moving towards laboratory exams from take-home exercises or blended assessment does not appear to reduce natural similarity. Given that less constrained assessments tend to be thought of as more valid, this weakens the case for laboratory exams in programming. Also, given that the nature of laboratory exams is such that they may have to be run over several sessions, the reliability gained may not be as much as might be hoped — our results do indicate that later sessions of the same laboratory exam have higher similarities. However, given the apparently strong effect of divergent assessment, and the likelihood of constrained exams being less convergent, it is hard to separate the two effects with a small number of assignments.

- There is some evidence, although not strong, that randomisation between individual students reduces similarity, possibly reducing overall similarity over multiple laboratory exam sessions to the same level as that found in the first session of a laboratory exam with randomisation between sessions. Possibly a bigger advantage of this approach is the possibility of deducing which student is behind an anonymous request for contract cheating, as opposed to in-cohort plagiarism.

- Specifying Java interfaces doesn't appear to have much impact on natural similarity. Given the utility of interfaces in automated testing/assessment, this is a positive result in favour of interfaces.

4.2 Implications for Assessment and Feedback

While the blending of assessment between constrained and unconstrained work appears to offer more validity with the possibility of better reliability, the other type of blended assessment that we have discussed, i.e. blending of automated and manual assessment, appears to work against reliability with a possible improvement in validity. However, there are other factors to consider in assessment, particularly the quality of the feedback given to students as to how their work could be improved. We have discussed ways of automating not only the process of assigning a mark to a piece of work, but also relevant comments.

Automated assessment is often used to give immediate feedback to students on whether they have got the (or a) correct answer without intervention from the teacher. Our experience has been that, unless multiple submissions are allowed, this leads to problems with validity because a student may have got everything right except perhaps one constructor. If that constructor is used in the automated assessment then nothing works and the student gets zero marks and fairly unhelpful feedback comments. This kind of assessment also contributes to the bimodal distributions that are often encountered in the distribution of marks in programming exercises [7]. By identifying the problem cases and allowing the assessor to make modifications to submissions (with appropriate commentary and allocation of marks), a more valid assessment can be built, at the cost of students having to wait longer for feedback. For instance, about one third of all submissions to the UGw12 assignment described above had some corrections applied to the code, whether automatically or manually.

4.3 Further work

We can identify many areas for further work based on the proposals and study that we have carried out.

- Work with a larger set of assignments to compare similarity scores, to deal with variation between institutions, and to allow the use of more sophisticated statistical tools to validate our conclusion that divergence vs convergence of assessment is the dominant factor.

- Use a larger data set to examine and unpick the apparently less significant effects due to blending and randomisation.

- Integrate more measures into the analysis, e.g. prescription of coding standards, and investigate the impact on similarity.

- Apply and compare other plagiarism detection tools to see what effect they have on similarity measures, if any.
- Examine the effect on similarity of different types of cohort i.e. novices vs more experienced programmers, undergraduate vs postgraduate.
- Look at assignments in other programming languages to how how similarity varies and how techniques for automated assessment carry across e.g. use of reflection for correcting code and/or identifying correct program structure; Java interfaces vs python abstract base classes.
- Seek student opinions about fairness of automatic and randomised assessment, and whether they understand the feedback.
- Investigate the effect of anti-plagiarism measures (e.g. constraining, randomisation, similarity tools) on students' inclination to plagiarise.

Although there are many areas for the work to be developed, the analysis of similarity measures appears to be a useful comparison to make, and reveals that randomisation and blended assessment may have positive impacts on the reliability of assessment, but divergent assessment is the most important factor that affects natural similarity, and hence the ease of detecting plagiarism, within an assignment.

5. REFERENCES

[1] M. Ardid, J. A. Gomez-Tejedor, J. M. Meseguer-Duenas, J. Riera, and A. Vidaurre. Online exams for blended assessment. Study of different application methodologies. *Computers & Education*, 81:296–303, Feb. 2015.

[2] J. Aycock. *Retrogame Archeology: Exploring Old Computer Games*. Springer, May 2016.

[3] D. J. Barnes and M. Kolling. *Objects First with Java: A Practical Introduction Using BlueJ*. Pearson, Boston, 5 edition edition, Sept. 2011.

[4] C. J. Bonk and C. R. Graham. *The Handbook of Blended Learning: Global Perspectives, Local Designs*. John Wiley & Sons, June 2012.

[5] K. W. Bowyer and L. O. Hall. Experience using "MOSS" to detect cheating on programming assignments. In *Frontiers in Education Conference, 1999. FIE '99. 29th Annual*, volume 3, pages 13B3/18–13B3/22 vol.3, Nov. 1999.

[6] V. Caravias. Literature Review in Conceptions and Approaches to Teaching Using Blended Learning. *Int. J. Innov. Digit. Econ.*, 6(3):46–73, July 2015.

[7] R. Cardell-Oliver. How Can Software Metrics Help Novice Programmers? In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, pages 55–62, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc.

[8] D. Chuda, P. Navrat, B. Kovacova, and P. Humay. The Issue of (Software) Plagiarism: A Student View. *IEEE Transactions on Education*, 55(1):22–28, Feb. 2012.

[9] G. Cosma and M. Joy. An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. *IEEE Transactions on Computers*, 61(3):379–394, Mar. 2012.

[10] Q. Cutts, D. Barnes, P. Bibby, J. Bown, V. Bush, P. Campbell, S. Fincher, S. Jamieson, and T. Jenkins. Laboratory exams in first programming courses. In *Proceedings of 7th Annual Conference of the ICS HE Academy*, pages 182–196, 2006.

[11] C. Domin, H. Pohl, and M. Krause. Improving Plagiarism Detection in Coding Assignments by Dynamic Removal of Common Ground. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '16, pages 1173–1179, New York, NY, USA, 2016. ACM.

[12] S. H. Edwards and M. A. Perez-Quinones. Web-CAT: Automatically Grading Programming Assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pages 328–328, New York, NY, USA, 2008. ACM.

[13] J. Hage, P. Rademaker, and N. v. Vugt. A comparison of plagiarism detection tools. Technical Report UU-CS-2010-015, Department of Information and Computing Sciences, Utrecht University, 2010.

[14] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppala. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

[15] A. Irons. Reducing plagiarism in computing. In A. Irons and S. Alexander, editors, *Effective learning and teaching in computing*, Effective learning and teaching in higher education series, pages 100–110. RoutledgeFalmer, London, 2004.

[16] M. Joy and M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, May 1999.

[17] V. Karavirta, A. Korhonen, and L. Malmi. On the use of resubmissions in automatic assessment systems. *Computer Science Education*, 16(3):229–240, Sept. 2006.

[18] M. Llamas-Nistal, M. J. Fernandez-Iglesias, J. Gonzalez-Tato, and F. A. Mikic-Fonte. Blended e-assessment: Migrating classical exams to the digital world. *Computers & Education*, 62:72–87, Mar. 2013.

[19] M. McAlpine. *Principles of assessment*. CAA Centre, University of Luton, 2002.

[20] E. O'Loughlin and S. J. Osterlind. A Study of Blended Assessment Techniques in On-line Testing. NUI Maynooth, 2007.

[21] L. Prechelt and G. Malpohl. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science*, 8(11), 2002.

[22] J. Sheard, Simon, A. Carbone, D. D'Souza, and M. Hamilton. Assessment of Programming: Pedagogical Foundations of Exams. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 141–146, New York, NY, USA, 2013. ACM.

[23] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, Sept. 2011.