Requirements for Author Verification in Electronic Computer Science Exams

Julia Opgen-Rhein[1], Bastian Küppers[1,2] and Ulrik Schroeder[2]
[1]*IT Center, RWTH Aachen University, Seffenter Weg 23, Aachen, Germany*
[2]*Learning Technologies Research Group, RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany*

Keywords: E-assessment, Cheating, De-anonymization, Stylometrics.

Abstract: Electronic exams are more and more adapted in institutions of higher education, but the problem of how to prevent cheating in those examinations is not yet solved. Electronic exams in theory allow for fraud beyond plagiarism and therefore require a possibility to detect impersonation and prohibited communication between the students on the spot and a-posteriori. This paper is an extension of our previous work on an application for detecting fraud attempts in electronic exams, in which we came to the conclusion that it is possible to extract features from source code submitted for tutorials and homework in a programming course. These can be used to train Random Forest, Linear Support Vector Machine, and Neural Network classifiers and assign the exams from the same course to their authors. The Proof of Concept was further developed and this paper outlines the experimentally determined requirements for the selection of training and test data and its pre-processing to achieve applicable results. We achieve an accuracy larger than 89% on a set of source code files from twelve students and found that material from all parts of a programming course is suitable for this approach as long as it provides enough instances for training and is free of code templates.

1 INTRODUCTION

Electronic exams are increasingly employed by institutions of higher education. Especially in programming courses, it is desirable to enable the students to be tested on their coding skills in a more realistic setting than a pen and paper exam. However, switching to another form of examination does not solve the problem that some students resort to cheating to obtain a better grade. At first appearance e-assessment seems to offer even more or at least some new possibilities to cheat (Heintz, 2017) - even more so, if the students are allowed to work on their own computer (bring your own device, BYOD). Cheating in an educational context can take many forms and ranges from illegal practices like stealing over collusion in assignments to copying (Sheard and Dick, 2012). Especially illegal collaboration and plagiarism seem easier in an electronic exam due to the possibility to exchange and obtain larger amounts of information and whole files via the internet or a Bluetooth connection.

The reasons for cheating are manifold and include disadvantageous personality traits (Williams et al., 2010) as well as practical reasons like a high workload, pressure and fear of failing (Sheard and Dick, 2012). This indicates that especially undergraduate students of computer science who struggle to learn programming are likely to cheat.

Although there are, unlike to online exams, lecturers and invigilators present during the examination, it cannot be completely ruled out that a cheating attempt is successful. The a-posteriori detection of cheating by eyesight of the lecturer is time consuming, uncertain and not feasible for large courses (Zobel, 2004). Additionally, it is hard to convict the offenders if they do not admit to cheating since the suspicion and intuition of the examiner is subjective and no reliable evidence. Conventional plagiarism detection software for written texts is not suitable for source code because of its formalized structure and fixed keywords. That is why a number of applications have been developed to particularly address plagiarism. These approaches all fail in cases where the source of the copied solution lies outside the reach of the examiner, e.g. code copied from the internet or a solution procured via communication with someone outside the examination room. Impersonation, that is another person taking the exam or submitting the solution under a false name, is also not detected by plagiarism detection tools.

Opgen-Rhein, J., Küppers, B. and Schroeder, U.
Requirements for Author Verification in Electronic Computer Science Exams.
DOI: 10.5220/0007736104320439
In *Proceedings of the 11th International Conference on Computer Supported Education (CSEDU 2019)*, pages 432-439
ISBN: 978-989-758-367-4

Therefore, a different way of cheating detection is required. One way to do this is using author verification methods to check the authorship of the person who hands in the exam. Instead of comparing the submissions to one another or to external sources, the exams of students are compared to their previously submitted solutions for similar tasks and assignments.

Previous work of the authors resulted in a prototypical application that uses machine learning techniques to verify the authorship of solutions in electronic exams (Opgen-Rhein et al., 2018). It can process Java source code files, which is still one of the most popular first programming languages students learn at university, along with C++ and Python (Aleksić and Ivanović, 2016), (Siegfried et al., 2016).

This approach depends on the hypothesis that even novice programmers have an individual programming style that can be used to distinguish their source code from the one developed by others. Programming styles are distinguishable even if the students are working on the same task and learn from the same script and tutor. These styles influence the layout of the code, the naming of classes, methods and variables and the use of different keywords and constructs of a programming language. There is no right or wrong in such stylistic choices and they are made independent from the tasks which students solve. Figure 1 illustrates this. The two methods differ in various aspects such as the number of spaces and blank lines, the accessibility of the method and the type of loop used for the summation.

This paper lists the conditions and data processing steps that are necessary to transform the raw submissions into a form suitable for the application to learn each student's programming style and achieve a feasible accuracy in the context of a programming exam.

```
public double calc_pi(int it) {
        double pi = 0;

        for (int i = 1; i < it; i++) {
            double r = 1.0 / (2.0 * i - 1);
            if (i % 2 == 0) {
                r *= -1.0;
            }
            pi += r;
        }
        return pi * 4.0;
    }
```

```
private double calcPi(int it){
        double pi=0;
        int k=1;
        while(k!=it){
            if(k%2==0)
            {
                pi+=-1.0/(2.0*k-1);
            }
            else
            {
                pi+=1.0/(2.0*k-1);
            }

            k++;
        }

        return 4.0*pi;
    }
```

Figure 1: Two different solutions to the task 'Write a method that calculates π with the Leibniz formula and stops the summation after *it* iterations'.

2 RELATED WORK

In the context of an exam, different kinds of cheating can occur. One is plagiarism, i.e. that a student copies parts of or the whole answer to a question or assignment from another student or an external source. This can be discovered by using plagiarism detection tools for source code, if the source of the duplicated solution is known so that a comparison can be made. Plagiarism detection is independent from the skill level of the student and the type of task although for programming tasks, a different approach has to be chosen than for natural language texts. If two exams are similar according to such a tool it cannot be determined, if they both had access to the same (internet) resource or *if* one of them and who gave the solution to the other student.

The TeSLA Projekt (Noguera et al., 2017) aims at developing a trust-based system for e-assessment in online learning scenarios. The authors describe various measures to prevent and detect cheating, including biometric recognition for authentication, plagiarism detection and forensic linguistics techniques. A solution for submissions in other forms them natural language is not mentioned.

The plagiarism detection tool MOSS has already been successfully used to find similarities between sections of source code (Schleimer et al., 2003). It finds similar sequences of strings by local fingerprinting. MOSS is able to perform fast comparisons between different kinds of sources, so that a comparison with internet resources is possible, but this system can be deceived by renaming of variables and methods.

A slightly different approach is followed by the authors of the program JPLAG (Malpohl et al., 2002). They break down the source code into tokens and search for similar sequences of maximal length of tokens between files. But the problem remains that the plagiarism detection only works in cases where the source is available for comparison. Moreover, the tasks that are solved in introductory programming courses usually result in code that does not differ much in structure, e.g. all students write a class with certain attributes and methods.

Earlier work by Peltola et al. does not use features derived from the source code but utilizes the fact that students write their texts on electronic devices. They use an analysis of recorded keystrokes during the solution of exercises and the exam for author identification (Peltola et al., 2017). This works for both programming and writing tasks and is independent of *what* is written. The disadvantage of this approach is that additional software is needed to track typing behavior on students' devices.

A study from (Caliskan-Islam et al., 2015) extracts features from C++ source code and uses a random forest classifier for author attribution. This approach was modified for Java code. We were able to show that such an approach is promising even though the results are weaker for programming novices than for experienced programmers. This technique is suitable for cheating detection because there is usually source material available from every student in the form of assignments and tutorials. Instead of comparing submissions from different students, a 'positive' comparison is made and the question that is answered is 'was the code written by this student?'.

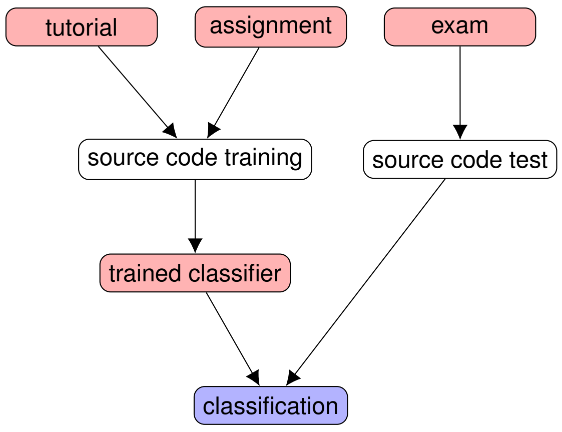

Figure 2: Workflow of the Application.

Table 1: Structure of the Feature Vector.

number of chars
average line length
standard deviation of line length
number of lines
[term frequency of all words]
frequency of 'else if '
max node depth in AST
[TF, TFIDF, avg node depth of AST node types]
[term frequency of AST node bigrams]
syntactic features
semantic features

2.1 Previous Work

Previous research led to the development of a prototype application that extracts features from Java code files, trains random forest, support vector machine and neural network classifiers and attributes unknown Java source code to one of the authors from the training set with a good accuracy. In this setting, the random forest classifier performed best with an accuracy around 58.36% on a small data set of 12 students. The feature set is large which leads to a high dimensional feature space and long computing times for bigger data sets. This paper describes the improvements that can be achieved by applying better data selection, feature reduction techniques and the combination of all three types of classifiers into an ensemble.

2.1.1 Features

For each source code file a feature vector is constructed. The feature set is an adaptation of the Code Stylometry Feature Set proposed by (Caliskan-Islam et al., 2015) and consists of the components listed in Table 1.

The first entries of the feature vector describe the layout of the code and can be computed by treating the source code as a plain text file. The parts of the vector in square brackets have a variable length which depends on the amount of distinct (key) words and constructs the programmer uses and therefore correlates with the file length. Syntactic features capture design decisions like the placement of brackets, the use of tabs and the number and type of comments. Semantic features describe the structure and inner logic of the code, e.g. the type of loop that is used, the accessibility of classes and variables or the number of classes and enumerations per file. The application computes 25 syntactic and 426 semantic features. Table 2 shows examples of them.

Other machine learning approaches for author verification and also the plagiarism detection tools do not consider the layout of the code and focus solely on its structure. Since beginners lack knowledge of many assets of a programming language their results differ less than the ones of more experienced authors which makes the utilization of the additional layout features promising.

The computation of the features is done with two techniques: The layout features, syntactic features as well as the term frequency (TF) and term frequency - inverse document frequency (TFID) are extracted by

Table 2: Examples of Syntactic and Semantic Features.

Feature	Category
# spaces	Syntactic
# empty lines	Syntactic
max # consecutive empty lines	Syntactic
avg # consecutive empty lines	Syntactic
# { in new line	Syntactic
# { in same line	Syntactic
# short notations (e.g. +=)	Syntactic
# inline comments	Syntactic
# block comments	Syntactic
# consecutive calls (e.g. do1().do2())	Semantic
# classes	Semantic
# chars/variable name	Semantic
# underscores/method name	Semantic

treating the source code as text and applying regular expressions. All other features are calculated by using an abstract syntax tree (AST). It is constructed using the Python library *javalang*[1] and breaks down the source code into tokens and nodes that describe its structure. The features that are directly computed from the graph are an indicator for the complexity of the code. Node bigrams are pairs of nodes that appear one after another, their frequency hints to a preference of certain constructs e.g. the invocation of the constructor of a super class. The feature vectors are scaled to a maximum value of 1 because most classification algorithms other than the random forest require normalized data to work correctly.

2.1.2 Classification

The samples are classified independently by three different algorithms: Random forest classifier (RF), linear support vector machine (SVM) and a neural network (NN). The application uses implementations of these classifiers from the *scikit-learn*[2] library which makes is possible to obtain the probability with which a sample is attributed to a class and hence the certainty of the decision. This is utilized for weighing of the algorithms in case they come to different results. Additionally the most important features can be derived so that it can be indicated on which properties the decision is based. The three machine learning algorithms can either be trained to distinguish between all authors (multi class classification) or to distinguish between one author and all others (binary classification). Multi class classification is a case of *author identification* since it presumes that the actual author is present in the training set while binary classification aims to decide if a piece of code was written by a specific author. If a student received a solution from someone who took the same exam, multi class classification will attribute this solution to its real author. In binary classification the sample is attributed to the class with the highest probability. For this approach, a classifier has to be trained for each student and the imbalance of classes represented in the training set has to be considered. If the probability for a class is smaller than 50%, this means that the algorithm attributed a sample to another author, but in cases where it is smaller for *all* classes, the class with the highest probability is assumed. Both approaches are implemented in the application and the parameters of the algorithms were optimized separately for both cases.

[1]https://github.com/c2nes/javalang

[2]https://scikit-learn.org/stable/

3 CONDITIONS

This paper analyses the conditions that influence the accuracy of the classification. Generally, the learning process of a ML algorithm depends on the amount and quality of the training material. Particularly in the setting of cheating detection for programming exercises this means that it has to be similar in structure to the actual exam and must be authored by the student alone.

The necessary amount of reference material can be examined by varying the number of samples in the training set. A programming course during the semester at university level usually consists of a lecture that is accompanied by a tutorial and possible assignment that students solve at home and hand in for grading. In this setting it is possible to gain at least one file per week form each student. Group work and collaboration of students is usually desired, but it should be taken care, that each submission is written by a single student and no group work is handed in. In general, as much training material as possible should be gathered. The similarity to the exam should be automatically fulfilled if the reference material stems from the course that is completed with the exam. The last requirement is the hardest to guarantee because tutorials are obviously not as strictly invigilated as exams and take-home-assignments are not checked at all. On the other hand it should be considered that it is rather unlikely that students hand in solutions that are not their own from the beginning instead of trying to solve them themselves.

Other important requirements for the training and test data include:

1. The source files are long enough to show the coding style

2. The reference material is not too old, i.e. belongs to the actual course, in case the students change their style due to increased aptitude and knowledge of new language features
3. The code is written with the same IDE and in the same programming language
4. The code compiles so that an AST can be constructed

The use case for this approach is a BYOD device exam of a programming course that can be open book in the sense that it allows the students to have access to their own code that they created during the semester in tutorials or during practicing. This setting fulfills most of the conditions automatically because the exam is done on the same device and most likely with the same IDE as the previous assignments.

Of course, a program alone cannot decide whether an exam is a case of fraud, since plagiarism is a serious case of academic fraud that can result in legal prosecution (Zobel, 2004). If a software detects that the work of a student does not resemble the previous submissions, this should be considered a hint for the examiner, not a final decision.

4 METHODOLOGY

To determine the necessary properties of the training and test data, tests were carried out using the previously developed application. The experiments are designed to answer the following questions:

1. What kind of source code is applicable for cheating detection and how does it have to be processed to construct the feature vectors?
2. What level of complexity in the data set is needed to obtain good results?
3. What features are useful? Is it possible and beneficial to reduce the complexity of the feature vectors?

The tests were performed with a set of assignments from an introductory Java programming course. The set contains the solutions to 12 on-campus tutorials from twelve students. All students solved the same task individually on their own laptop. They handed in between one and five source files per task without any requirements concerning the file structure of the solution. The length of the files varies between 8 and 279 lines. In most cases, snippets for code testing were copied and pasted from the description. This corpus consists of 437 Java source code files with an average length of 41.69 lines.

Additionally, the submissions of the best 40 participants of the 2017 Google Code Jam (GCJ) programming contest[3] was used for comparison. The files from the second data set on average contain more lines and the solution to each task is contained in a single file while the students often handed in more than one file and additional test classes. This data set represents a high level of complexity and was written by experienced programmers. This data set is comprised of 699 files with an average length of 160.49 lines.

5 REQUIREMENTS

5.1 Data Selection

The number of lines of code that is sufficient for classification is hard to determine, so that it was decided to keep all files, regardless of their length. Caliskan et al. state that just 8 training files are sufficient to classify a large number of authors correctly (Caliskan-Islam et al., 2015), but they tested their approach on files that consist of significantly more lines of code and contain a whole program, while students are often advised to distribute their code on multiple files containing one class each. Files that do not consist of valid Java code or do not compile are discarded by the application because no feature vector can be computed from them. Especially in a beginner's course the first lectures can consist of getting familiar with the IDE and producing simple text output by copying some kind of 'hello world' program. Such files are omitted. Since there are no exams from the course available, the files are randomly split into training and test data via cross validation. Files that are not written by the students themselves like test classes are removed manually from the data set. This resulted in 47 of 437 files from the raw data set not being used for classification.

The data from the second data set was simply saved in a folder structure that can be processed by the application and all Java submissions from the contestants were used.

5.2 Pre-processing

After computing the feature vectors, outlier detection is performed to obtain a less noisy data set. The removal of duplicate (test) files is done by clustering the data and removing files with the same name if they form a cluster for all the students.

[3]https://code.google.com/codejam/past-contests

After reading and processing all samples it has to be checked whether there is a sufficient amount of (or at all) training data present for the classes in the test set. Otherwise, no classification is performed and the exam has to be checked with another method.

Since large feature vectors lead to a high computing time for the application, feature reduction techniques were tested. Principal Component Analysis (PCA) was done on the training set and the 250 features that contributed most to the variance were kept. These features kept around 90% of the overall variance. The same reduction was then done on the set of samples that needed to be attributed. PCA was beneficial for the results of SVM but reduced the accuracy of RF and NN classifiers. The reason for that lies in the nature of these classifiers. A reduction of dimensionality makes it easier for the SVM to separate the data while it removes features that the RF classifier needs in a split for the decision. Fewer features mean that the Random Forest consists of a lower number of different trees.

6 RESULTS

By the steps described in the previous sections it was possible to increase the recognition rates on the data sets significantly in comparison to the old approach (Opgen-Rhein et al., 2018). To assess the influence of the data selection and pre-processing, the accuracy of the three classifiers was calculated as the average after performing ten times 10-fold cross validation. Table 3 shows the final results for each classifier for multi class and binary classification.

Table 3: Accuracy of NN, SVM and RF using 10-fold cross validation - small data set.

	RF	lin. SVM	NN
multi	89.15%	77.18%	78.49%
binary	88.72%	80.15%	78.85%

Table 4: Accuracy of NN, SVM and RF using 10-fold cross validation - Google Code Jam data set.

	RF	lin. SVM	NN
multi	100.00%	99.34%	99.69%
binary	99.41%	100.00%	99.81%

As expected, a sufficient amount of data (at least ten files as training data from each student) and a number of snippets with a sufficient length of the code is an essential prerequisite for a successful classification. The results above were obtained by using all available code for training. The low number of training files described by Caliskan et al. only works on the Google Code Jam Data set. This is due to the greater differences between the tasks for students who range from writing classes, interfaces and exceptions to recursion and often focus on one topic per task, while the contest asks for reading a text file, processing the data and writing the result to a text file in all cases.

Due to the assignment description, the students often turned in test cases that were copied from the description to show that their code is functional. Test cases can either be written by the students themselves or are done in a separate file that is not used for classification. Dilemma: own testing has a learning effect but is harder to assess and grade.

The analysis of the most important features for the data sets made it clear, that it is not possible to find a small feature set that can be used to discriminate between all authors. The features that contribute most to the decision vary from data set to data set and between runs and classifiers. Therefore, it is necessary to compute a big feature set over all training examples and reduce it afterwards, depending on its inner variance. With respect to the application's run time this is feasible since the computation of the feature values and their reduction is not computationally expensive in comparison with the actual classification.

The results of multi class and binary classification show no big differences. Binary classification yields better classification results for the linear SVM while multi class classification works better for the RF classifier. Multi class classification should be preferred because it is evidently less time consuming to train one classifier to distinguish between *n* classes than to train *n* classifiers to distinguish between one author and all others.

The files from the student data set that were misclassified turned out to be mostly very short files like interface declarations and custom exceptions that are too simple to show the individual coding style. These files *can* remain in the training set but are useless if they are the only sample that is used to check the authorship.

Additionally, homework for which the teacher provided parts of the result, for example a code skeleton, is more often confused and should not be used. This constraint must hold for the exam as well.

The results show that pure 'data containers' with pre-defined names for classes and methods are also less useful for classification.

Finally, a weighting of the algorithms is needed, too. Since the analysis of the classifications showed that in many cases the real class of a sample did not have the highest probability but appeared in second or third place. It is assumed that there has been no fraud, if at least one of the following conditions holds for one of the classifiers:

- The classifier gives the highest probability to the class of the presumed author
- The probability of the class of the presumed author is bigger than 50%, even if it is not the highest probability (this is possible in case of binary classification)
- The probability of the class of the presumed author is the second or third highest probability but amounts to at least 0.8 times the highest probability

This is especially beneficial for the linear SVM classifier because it was found to return its results with a lower certainty than the other two. The results for this relaxed classification on the student data set are shown in Table 5. If the authorship of a student is considered to be verified, if one of the algorithms confirms it (comb.), the number of cases that have to be checked for fraud by a human examiner is reduced by more than 90%. For the GCJ data this technique resulted correct author attribution in 100% of the cases.

Table 5: Accuracy of relaxed classification for the student data set using ten times 10-fold cross validation.

	RF	lin. SVM	NN	comb.
multi	91.18%	81.64%	79.10%	91.46%
binary	90.44%	83.54%	80.41%	90.95%

7 DISCUSSION AND FUTURE WORK

Data selection and pre-processing are crucial steps to obtain data for cheating detection. Their usefulness was tested by looking at changes in the classification accuracy.

From our findings, rules for the construction of assignment tasks and electronic examinations that shall be checked for cheating can be derived. These conditions lead to a number of rules that must be obliged for assignments, tutorials and exams in order to allow classification for cheating detection afterwards: the students have to submit their own work in all cases and the number of assignments should produce at least ten source files with testing in an extra file (which is good practice anyway, especially if testing is done with unit tests). The results show that pure 'data containers' with pre-defined names for classes and methods are not useful for classification. The ideal exam for this scenario asks the students to apply the things they learned in class and practiced at home and during tutorials. Its length might be higher than the length of an assignment because it asks to activate knowledge rather than forcing to explore something new. In an open book exam it is even allowed to copy and paste code that a student himself wrote earlier. Although the exam presents a 'mix' of assignment types the approach still yield satisfactory results. The exam should be (as it is in most cases and in paper based exams, too) subdivided thematically into tasks. The solution of each task should be written into one source file. More than one file for testing also offers the chance to rule out impersonation (but not plagiarism) in cases where not all but at least the majority of files of one submission are attributed to the student who claims to have written it.

It is beneficial for the classification if the code samples reflect the programming style of the student alone and is free of parts that are written be e.g. the lecturer as it might happen if examples and test cases that are provided.

While a certain degree of similarity (like the same task and same file processing scheme in GCJ solutions) between the samples is useful for the classifiers to find subtle differences in the source code files, greater differences are harmful for classification. It is best if the samples contain the same task for all the students.

If the same tasks lead to the training examples this is not a problem: the classifiers will concentrate more on the subtle differences between the authors than on e.g. the naturally occurring differences between front end and a back end code.

The data used comes from very formalized tasks. A new course design should focus on giving students more choice in the naming and structure of their code and more focus on functionality. At the same time, the number of training instances can be increased by ending the practice of group homework or requiring stating the author for each file.

Regarding the research questions it can be concluded that code from all parts of a programming course can be used for cheating detection as long as it can be reasonably assumed that it is indeed authored by one particular student. The code must compile and be free of testing snippets that are used by all students, i.e. testing needs to be done in separate files that can easily be removed from the training set. A minimum number of ten reference files are required but the accuracy increases with more training material.

To obtain usable results the source code does not need to be particularly complex as long as it is long enough. If all programmers work on a similar task or all tasks have the same structure (as it is the case in the GCJ contest) this is even beneficial because the classification is focused on the differences in coding instead of the structure of the task. Still, a certain complexity is needed in the sense that it must be possible to express one's own coding style. Simple output of data and pure 'data container' classes should be avoided.

The usefulness of the features depends on the data set and it is not possible to construct a reduced feature set that is useful for all data sets. But it is possible and useful to reduce the features after computation for the whole training by removing features with low variance and zero mutual information. Mutual information describes the relation between two data sets, in this case between each column (feature) in the training set and the label vector y (Ross, 2014).

Finally, it should be noted that all students must agree that their tasks will be used for fraud detection. This agreement can be obtained when students upload their work.

Further research is necessary to rule out that programmers are assumed to be the authors of files although they are not, i.e. a case of cheating is not detected by the application. If the training set is big, it is unlikely that this happens if it is assumed that in case the real author is not in the training set, the file will be randomly attributed to one of the classes. But this will most likely not be the case, especially if the students have knowledge about the cheating detection process. This might cause them to obtain code from a person which they think has a coding style that resembles theirs.

The next step is the integration of the cheating detection into an e-assessment framework. This also includes taking into account measurements that prevent a cheating attempt while the exam takes place. Since the first electronic examinations took place attempts have been made to secure the computers on which the exam is taken and prevent the use of forbidden software and the internet. This is relatively easy for workstations that are under the control of the examiners and whose configuration is known and homogenous (Wyer and Eisenbach, 2001).

REFERENCES

Aleksić, V. and Ivanović, M. (2016). Introductory programming subject in european higher education. *INFORMATICS IN EDUCATION*, 15(2):163–182.

Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., and Greenstadt, R. (2015). De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 255–270, Washington, D.C. USENIX Association.

Heintz, A. (2017). Cheating at Digital Exams. Master's thesis, Norwegian University of Science and Technology, Norway.

Malpohl, G., Prechelt, L., and Philippsen, M. (2002). Finding plagiarisms among a set of programs with jplag. *JUCS - Journal of Universal Computer Science*, 8(11).

Noguera, I., Guerrero-Roldán, A.-E., and Rodríguez, M. E. (2017). Assuring authorship and authentication across the e-assessment process. In *Technology Enhanced Assessment*, pages 86–92. Springer International Publishing.

Opgen-Rhein, J., Küppers, B., and Schroeder, U. (2018). An application to discover cheating in digital exams. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, Koli Calling '18, pages 20:1–20:5, New York, NY, USA. ACM.

Peltola, P., Kangas, V., Pirttinen, N., Nygren, H., and Leinonen, J. (2017). Identification based on typing patterns between programming and free text. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Calling '17, pages 163–167, New York, NY, USA. ACM.

Ross, B. C. (2014). Mutual information between discrete and continuous data sets. *PLoS ONE*, 9(2):e87357.

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA. ACM.

Sheard, J. and Dick, M. (2012). Directions and dimensions in managing cheating and plagiarism of it students. In *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*, ACE '12, pages 177–186, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Siegfried, R. M., Siegfried, J. P., and Alexandro, G. (2016). A longitudinal analysis of the reid list of first programming languages. *Information Systems Education Journal*, 14(6):47–54.

Williams, K. M., Nathanson, C., and Paulhus, D. L. (2010). Identifying and profiling scholastic cheaters: Their personality, cognitive ability, and motivation. *Journal of Experimental Psychology: Applied*, 16(3):293–307.

Wyer, M. and Eisenbach, S. (2001). Lexis: an exam invigilation system. In *Proceedings of the Fiftteenth Systems Administration Conference (LISA XV) (USENIX Association: Berkeley, CA, p199, 2001. International Conference on Engineering Education August 18–21, 2002*.

Zobel, J. (2004). "uni cheats racket": A case study in plagiarism investigation. In *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30*, ACE '04, pages 357–365, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.