Automatic Programming Error Class Identification with Code Plagiarism-Based Clustering

Sébastien Combéfis
École Centrale des Arts et Métiers
Promenade de l'Alma 50
1200 Woluwé-Saint-Lambert, Belgium
s.combefis@ecam.be

Arnaud Schils
Université catholique de Louvain
Place de l'Université 1
1348 Louvain-la-Neuve, Belgium
arnaud.schils@gmail.com

ABSTRACT

Online platforms to learn programming are very popular nowadays. These platforms must automatically assess codes submitted by the learners and must provide good quality feedbacks in order to support their learning. Classical techniques to produce useful feedbacks include using unit testing frameworks to perform systematic functional tests of the submitted codes or using code quality assessment tools. This paper explores how to automatically identify error classes by clustering a set of submitted codes, using code plagiarism detection tools to measure the similarity between the codes. The proposed approach and analysis framework are presented in the paper, along with a first experiment using the Code Hunt dataset.

CCS Concepts

•Information systems → Clustering and classification; •Social and professional topics → Computing education; •Software and its engineering → Software defect analysis;

Keywords

Code Similarity; Automatic Code Assessment; Education

1. INTRODUCTION

Automatic programming codes assessment is a very demanded feature for the numerous online platforms to learn how to code and program [17]. In particular, the apparition of *Massive Open Online Courses* (MOOCs) increased the demand for such automatic assessments, in particular for the courses that ask the learners to produce code. They can therefore scale and reach a massive public [2, 22].

Being able to automatically assess codes is important, in particular to provide a grade to learners and to establish whether they succeeded the course or not. As argued in [5], another important part of the assessment is the feedback received by the learners, especially when the main goal is to actually help them to learn. Feedbacks used for education purposes and targeted to learners are very different from those targeted to developers and should be well designed.

This paper builds upon a previous work that developed an intelligent unit testing-based code grader for education [3, 4]. Exercises designed for that grader are embedding customised feedbacks output when codes written by learners fail to pass associated test cases. However, these feedbacks are associated to test cases that are hardcoded in advance. Moreover, they do not cover all the potential errors that could be made. Therefore, failing submissions sometimes do not have any other feedback than the results of the failing test case, which is more a developer feedback than a learner one. This paper proposes a framework to analyse existing submitted code in order to automatically identify error classes. The teacher can then attach one specific feedback message for each class of errors, increasing dramatically the covered potential errors made by learners.

1.1 Motivation

Providing a good feedback to learners is very important to support their learning. It is even more important when it comes to automatically assess codes in the frame of a learning platform, where the learner is not in front of a teacher. Helping teachers to understand learners difficulties and providing them with a global overview of the submitted codes is also important.

There exists a lot of different possible aspects of a program that can be assessed ranging from testing the code to checking its style [22]. Another semi-automatic possibility is code peer-review where the learners have to review codes written by their peers, under the supervision of the teacher [24, 21].

It is impossible to design an automated assessment system that can anticipate all the possible errors that a learner can do; moreover is it not possible to automatically provide feedbacks that will help for sure the learner to understand the failure and fix its code in all situations. However, for introductory courses, learners are very often making the same mistakes, and the produced codes are generally shorts. It is therefore easier to get a better coverage of all the possible error classes.

All these observations motivated the work presented in this paper. More precisely, this work has two goals:

1. Given a set of codes, help teachers to identify the main error classes produced by the learners.
2. Given one code that fails some tests, generates a good quality feedback that helps learners to understand the failure and correct it.

CHESE'16, November 14, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4402-9/16/11...$15.00
http://dx.doi.org/10.1145/2993270.2993271

The first goal is an *a posteriori* analysis that helps teachers to understand the difficulties of learners and the errors they made. The second goal is an *on-the-fly* analysis of a submitted code that identifies the best suitable feedback to provide to learners.

1.2 Related Work

Various techniques and tools to automatically assess codes have been developed. Recent reviews [6, 10] highlight the diversity of the generated assessments and feedbacks that are returned to learners. Most of the assessments are generated through unit testing or other kinds of tests including acceptance tests, for example. The integration of manual assessments is also a common feature, with direct corrections by the teacher or code peer-reviews made by other learners.

Techniques relying on the measure of code similarity have been used to automatically produce code assessments. For example, semantic similarities between a submitted code and correct program models are computed to evaluate how close the submission is from the correct solution [15, 23]. That information is then used to generate a relevant feedback to help learners reaching a correct solution. Singh et al. [20] and Glassman et al. [8, 7] also worked on the identification of similar code submissions.

The remainder of the paper is organised as follows. Section 2 first presents how the proposed approach uses code plagiarism detection tools to measure code similarity. It then presents the proposed analysis framework and how to analyse a set of submitted codes in order to be able to generate relevant feedbacks for a new code submission. Section 3 then describes some first experiments that have been performed and discusses the obtained results. It lays out lessons that can be learned from those results. Finally, the last section concludes the paper with future work.

2. ERROR CLASS IDENTIFICATION

The framework proposed in this paper relies on the ability to measure if two given codes are similar. This work uses code plagiarism detection tools to measure code similarity. This section first briefly presents how code plagiarism detection can be achieved. It then presents the proposed approach used to automatically identify error classes.

2.1 Code Similarity

Measuring the similarity of two codes can be done in several ways as summarised in [16]. Simplest approaches compare codes just considering them as strings while more advanced approaches use lexers to compare sequences of tokens or parsers to compare abstract syntax trees (AST). The former approaches are simple and generic while the latter ones are language-dependent, but give a more precise similarity measure for code comparison. The main issue with approaches relying on the AST is that they require the code to compile to have a full AST.

Various tools have been developed to detect code plagiarism using code similarity [9, 13], among which the most famous are MOSS, JPlag and Sherlock. MOSS [19] supports the largest number of programming languages and works as a webservice. It is not open source and therefore only extendable by its authors. It computes fingerprints from the codes and then compare them using an efficient algorithm called *winnowing*. JPlag [18] only supports a few languages (Java, C#, C, C++, Scheme) and is released as a standalone application. It converts codes into token strings and then performs the comparisons with an algorithm called *greedy string tiling*. Sherlock [11] is an open source software that supports most procedural and object-oriented languages, with optimisations made for Java. Upgrading the tool to a new programming language is made easy. The comparison of codes is made incrementally by comparing them directly, with whitespace and comments removed and after tokenisation.

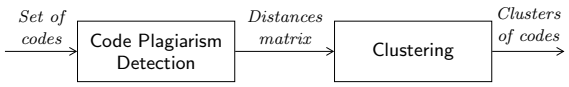

Figure 1: A code plagiarism detection tool is run on the set of submitted codes to compute the pairwise distances between codes. A clustering algorithm is then executed on the distances matrix to identify clusters of similar codes.

Code plagiarism detection tools can output different kinds of information as a result of the analysis. Given a set of programs, they typically output the percentage of similar code for each pair of programs. They can also propose a finer analysis and show two programs side-by-side, identifying the similar code chunks.

2.2 Analysis Framework

The first goal of the analysis framework is to make an *a posteriori* analysis of a set of submitted codes, to identify the main error classes that have been produced by the learners.

Figure 1 shows a general overview of the analysis framework. The main input is a set of codes that have been submitted by the learners for the same exercise. Ideally, only submitted codes that are not correct must be included in the analysis. The correctness can, for example, be assessed using a unit testing approach such as the one used in previous work [4]. The analysis then proceeds in two phases:

1. First, all the codes are analysed by the code plagiarism detection tool, which has been beforehand properly configured. The output produced by the tool is then parsed to extract the information necessary to build a distances matrix. This matrix contains a dissimilarity value for each pair of codes.
2. Then, a clustering algorithm is fed with the distances matrix and produces as output a set of clusters. The codes belonging to the same cluster are close from each other, and are distant from the code from the other clusters.

The teacher receives a set of clusters as a result of the analysis. Each cluster produced by the framework contains similar codes and is represented by a *"central"* member of the cluster. This work makes the hypothesis that each obtained cluster should represent one error class.

Clustering algorithms can be configured and could therefore produce different sets of clusters as a result. Choosing and adjusting these parameters can be done in two ways. The analysis framework can either try different configurations, choosing the one maximising an objective function evaluated on the produced set of clusters, or it can propose to the teacher several results and let him/her choose the most relevant one.

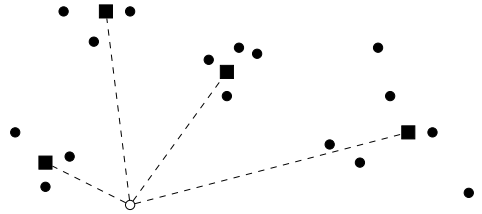

Figure 2: Each of the four detected clusters of codes (•) is represented by a "central" element (■). The feedback generated for a new submitted code (○) could be the one of the nearest cluster representatives.

2.3 Feedback Generation

Once a set of submitted codes has been analysed, the teacher has a set of clusters, with one representative by cluster as shown on Figure 2. The teacher has then to associate one feedback for each representative, that it relevant for the error class, that is, for all the codes in the same cluster.

Once the teacher has annotated the clusters, it can be used to provide feedbacks for new submissions. The distances between the newly submitted code and the representatives of the clusters are computed. The feedback of the nearest representative can then be used for the new submission if the distance is smaller than a given configurable threshold. This threshold has to be properly chosen to avoid false positive and the generation of a feedback that will surprise the learner since it may not be relevant enough.

3. EXPERIMENTS

A prototype of the analysis framework has been implemented in order to make some experiments. Codes used for the experiment have been extracted from the *Code Hunt dataset* [1]. Code Hunt [1] is a serious gaming platform, developed by Microsoft Research, that proposes coding contests helping their users to practice their programming skills. Puzzles are presented with test cases only and the user has to find the code that will pass the tests. The codes can be written either in C# or in Java. Hints are provided if the user cannot guess the correct code.

3.1 Prototype

The framework is developed with the R programming language [2]. This choice has been made to benefit from the numerous data mining algorithms it provides. Moreover, the clustering algorithms provided in the R standard library are quite efficient. It was therefore not required to try to take advantage from more advanced frameworks running on top of clusters such as Hadoop or to use lower level programming languages such as C++ for optimisation purpose.

3.1.1 Distances Between the Codes

The code plagiarism detection tool that has been used is JPlag [18]. This choice has been made since the output it produces contains all the necessary information to compute the distances between all pairs of codes. Moreover, it supports both C# and Java, the languages accepted by Code Hunt. JPlag has been configured with the following options:

- `-l c#-1.2` selects C# as the programming language;
- `-t 1` selects the sensitivity for the comparisons, a lower value representing a greater sensitivity.

C# has been chosen because there are more codes with that language in the Code Hunt dataset. Setting the sensitivity to the minimum value, that is, having the greatest sensitivity, is required because the analysed codes are short. Otherwise, too many codes are considered as 100% similar, even if they significantly differ, leading to a poor clustering.

The output produced by JPlag has the following structure, where elements between * are specific to one experiment:

```
Language accepted: C# 1.2 Parser
Command line: -l c#-1.2 -t 1 *path to code files*
initialize ok
*n* submissions

Parsing Error in *file_1*:
  *file_1*: *error_name*
  ...
Parsing Error in *file_n*:
  *file_n*: *error_name*

*m* submissions parsed successfully!
*n-m* parser errors!

Comparing *file_1*-*file_2*: *similarity*
...
Comparing *file_i*-*file_j*: *similarity*
...
Comparing *file_n-1*-*file_m*: *similarity*
```

The first block provides information about the configuration and the initialisation of the analysis. The second block lists the compilation errors which are present in the codes. The last block contains the pairwise similarity values, a number between 0 and 100. A tuple $(file_i, file_j, similarity)$ is extracted from each line of this last block.

To execute a clustering algorithm, the similarity information has to be formatted as a distances matrix. The distance between two codes i and j is computed as follows:

$$dist(i,j) = max\_possible\_similarity - similarity$$

Since the similarity value output by JPlag is a percentage, the maximum possible distance is 100 in this case. An integer between 1 and n is associated to each code. The (i,j) element of the distances matrix therefore contains the distance between the codes i and j. Since $dist(i,j)$ is of course equals to $dist(j,i)$, i.e the distance matrix is symmetric, memory can be saved by only keeping the lower triangle part of the distances matrix. The diagonal can also be dropped since $dist(i,i) = 0$, which does not provide any interesting information for the clustering.

The size of the distances matrix can be the cause of performance issues. Indeed, data mining algorithms based on distances matrix have a $\mathcal{O}(n^2)$ spatial complexity. However, it is not an issue when dealing with the Code hunt dataset, which can be handled on a standard computer.

3.1.2 Clustering of the Codes

Two clustering algorithms have been tried in the experiments. They directly come from the R standard library.

The first one is the *k-medoids* algorithm [12]. It chooses a medoid as the centre of a cluster, that is, one of its members

[1]The codes that have been used for the experiments come from the *dataset 1* available on GitHub: https://github.com/Microsoft/Code-Hunt.

[2]Code of the prototype used for the experiments is available on GitHub: https://github.com/aschils/cpbc

to minimise the average dissimilarity to its other members. In comparison, the k-means algorithm chooses a centroid, which is not required to be a member of the cluster. This difference is important since the centre will be the representative of the error class of the cluster and must have an associated custom feedback. The second difference is that the k-medoids algorithm works with any distance function and is usually directly fed with a distances (dissimilarity) matrix. The k-means algorithm takes a dataset as input and computes itself the distance between two elements, typically with the Manhattan or Euclidean distance function.

The k-medoids algorithm requires the number of clusters to be chosen *a priori*. It has either to be chosen by the teacher before launching the analysis, or it can be automatically adjusted, by choosing one in a given range to optimise a function of interest. Finally, it can also be automatically selected increasing k until convergence of the reconstruction error. This error is the sum of the distances between an element and the medoid of its cluster, over all the elements. This work uses this last approach that proved to be effective.

The second clustering algorithm that has been tried is the *agglomerative hierarchical clustering* [14]. It starts with one cluster for each element. At each iteration, it finds the two closest clusters and merges them. This work uses the Ward's minimum variance method of the R standard library. This method favours compact and spherical clusters.

Hierarchical clustering offers several advantages compared to the k-medoids algorithm. First, the number of clusters should not be selected *a priori*. Then, it provides additional information since clusters are organized hierarchically. This hierarchical structure can be drawn as a dendrogram. This structure can be displayed to the teacher who can, using a top-down approach, expand the clusters into subclusters until the best classification is reached, regarding the teacher's criteria and the statement of the exercise.

3.2 Experiments

Experiments have been made with exercises of the `Sector4_Level6` of the Code Hunt dataset, containing 53 submissions written in C#. The resulting clusters for different values of k are available on the GitHub repository of the proposed tool, for the two algorithms, in the `Sector4_Level6` folder. In addition to these clusters, the repository also contains a manual clustering of the codes. This clustering has been used to provide a first assessment of the quality of the obtained clusters.

3.2.1 *k-medoids*

With $k = 4$, the classification is already fine enough to spot some trends in the analysed codes. The first cluster is composed of codes containing one class `Program` and one method `Puzzle`. The body of the methods are composed of either one or two instructions, each time the same ones. When increasing k to 10, members of the first cluster obtained with $k = 4$ are spread in two different clusters: one composed of the codes where the body of the `Puzzle` method has only one instruction and one for the codes with two instructions in the `Puzzle` method body.

All the codes that use the `switch` statement are grouped in the second cluster obtained with $k = 4$. Codes from the third cluster exhibits three main trends: they contain a fibonacci-related method, a `char` processing related method or are using both `if` and `for` statements. This third cluster is also correctly split in several smaller clusters when k is increased. The fourth cluster presents around seven different trends and should therefore be split in smaller clusters too.

Manual inspection of these first results shows that clustering a set of codes using a code plagiarism detection tool provides relevant information about the codes that could be used and interpreted by teachers. Figure 3 shows three codes coming from the same cluster obtained with the k-medoids algorithm with $k = 11$. The first code is the medoid of the cluster, that is, its representative. The second code is the nearest from the medoid, that is, the most similar one. The only difference with the medoid is the presence of a cast operator and parentheses around the casted expression in the first instruction of the `for` loop. Finally, the third code is the farthest from the medoid.

```
using System;

public class Program {
  public static string Puzzle(string s) {
    char[] x = s.ToCharArray();
    int f1 = 1, f2 = 1, t;
    for (int i = 0; i < s.Length; i++) {
      x[i] = (x[i] - 'a' + f2) % 26 + 'a';
      t = f1;
      f1 += f2; f1 %= 26;
      f2 = t;
    }
    return new string(x);
  }
}
```

```
using System;

public class Program {
  public static string Puzzle(string s) {
    char[] x = s.ToCharArray();
    int f1 = 1, f2 = 1, t;
    for (int i = 0; i < s.Length; i++) {
      x[i] = (char)((x[i] - 'a' + f2) % 26 + 'a');
      t = f1;
      f1 += f2; f1 %= 26;
      f2 = t;
    }
    return new string(x);
  }
}
```

```
using System;

public class Program {
  public static string Puzzle(string s) {
    char[] arr = s.ToCharArray();
    uint fibim2 = 0, fibim1 = 0, fibi = 1;
    for(int i=0;i<arr.Length;++i){
      uint newchar = fibi % 26;
      if(arr[i] + newchar > 'z')
        arr[i] = arr[i] + newchar - 'z' + 'a' - 1;
      else
        arr[i] = (char)(arr[i] + newchar);
      fibim2 = fibim1;
      fibim1 = fibi;
      fibi = fibim1 + fibim2;
    }
    return new string(arr);
  }
}
```

Figure 3: Codes coming from the same cluster (containing five elements) exhibit similarities. The first one is the medoid of the cluster, the second one is the nearest from it and the third one is the farthest.

3.2.2 Agglomerative Hierarchical Clustering

The chosen dataset has also been analysed with the agglomerative hierarchical clustering algorithm. Figure 4 shows

the obtained result presented as a dendrogram. The leaves are the 53 submissions and the attached labels are the number of the ideal cluster according to the manual classification. A first look at the dendrogram directly shows that the codes from the same ideal cluster are in the same subtrees of the dendrogram, meaning that the results are consistent with the manual clustering.

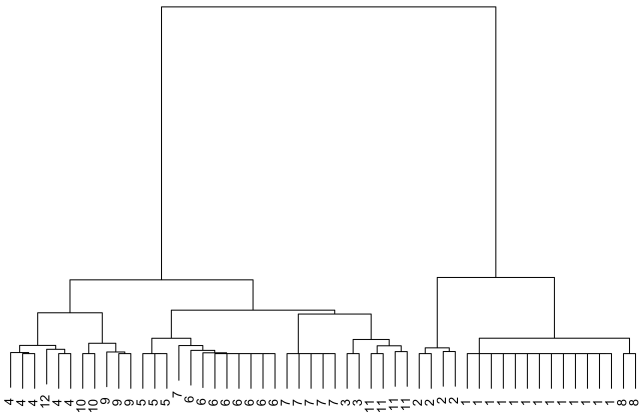

Figure 4: Result of the agglomerative hierarchical clustering algorithm as a dendrogram.

3.2.3 Evaluation

The quality of the obtained clusters with the two clustering algorithms has been realised using the manual clustering as a reference. The closer the obtained clustering is from the manual *"ideal"* clustering, the better its quality.

The score that is assigned to a given clustering is computed with Algorithm 1. It is a kind of similarity measure between the ideal clustering and the one computed by the proposed approaches. The score algorithm is designed so that the maximum possible score is 1.

Algorithm 1: Quality score evaluation of a given clustering compared to an ideal clustering.

Input: ds: the set of submissions.
Input: $c(x)$ and $i(x)$: two clusterings of ds as functions giving the cluster submission x belongs to.
Input: *ideal*: the ideal clustering of ds, that is, a set of submissions sets.

$score \leftarrow 0$
foreach $s_1 \in ds$ **do**
  $same \leftarrow \{s_2 \mid s_2 \in ds, c(s_2) = c(s_1), s_1 \neq s_2\}$
  $ideal \leftarrow \{s_2 \mid s_2 \in ds, i(s_2) = i(s_1), s_1 \neq s_2\}$
  $diff \leftarrow ds \setminus (ideal \cup \{s_1\})$
  $score \leftarrow score + \#(same \cap ideal)$
  $score \leftarrow score - \#(same \cap diff)$
$max\_score \leftarrow \sum_{j \in ideal} \#j \cdot (\#j - 1)$
return $score/max\_score$

The scores for the clusterings obtained with the k-medoids algorithm, for different values of k are plotted on the Figure 5. The maximum score, namely 0.9, is obtained for $k = 11$. Only two submissions amongst the 53 are misclassified in comparison to the ideal clustering. The first one is an outlier which is the only submission in its cluster in the ideal clustering, but that has been attached to a larger cluster by the proposed approach. The second one is just wrongly classified, according to the ideal clustering.

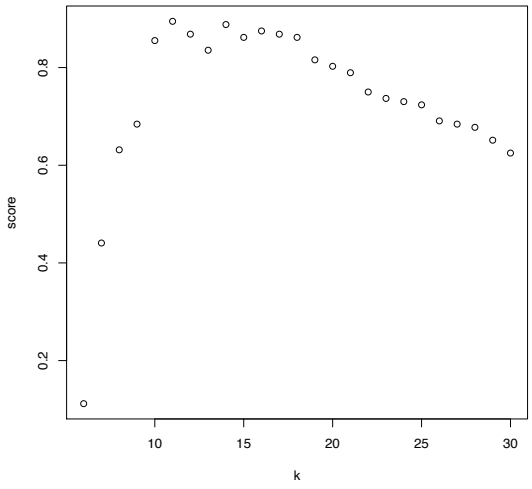

Figure 5: Quality scores for the k-medoids algorithm wrt the number of clusters k.

The agglomerative hierarchical algorithm also gives good results. Figure 6 shows the plot of the quality score of the clusterings obtained by cutting the dendrogram at different heights. The best score, namely 0.91, is for $h = 10$. The resulting clusters are slightly different from those output by k-medoid for $k = 11$.

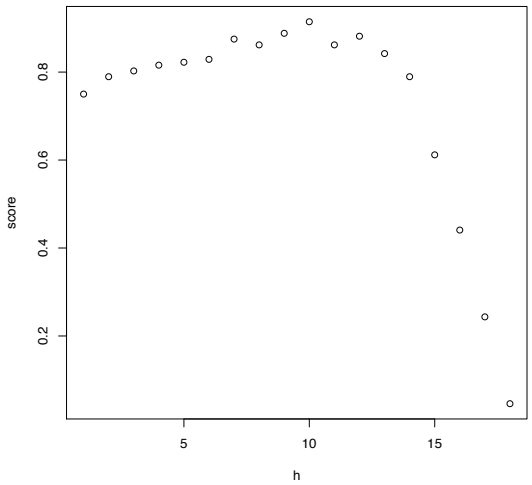

Figure 6: Quality scores for the hierarchical algorithm wrt the cut height h in the dendrogram.

The best clustering scores are nearly the same for the both clustering algorithms. Although additional experiments should be made, the proposed technique shows promising results provided that the k parameter is chosen properly. Allowing the teacher to test different values could be a nice feature to propose. The classification it achieves is very closed to the one that has been made manually.

4. CONCLUSIONS

To conclude, this paper proposes a framework to analyse a set of programming codes. The analysis is to be used by

teachers to help them to identify classes of errors that are made by learners. The proposed approach uses clustering algorithms measuring the similarity between pairs of codes with code plagiarism detection tools.

The proposed framework can be used on codes that fail to pass some tests, in order to find representatives of the error classes and assign them a custom feedback that helps the learner to understand and fix the failure. The failing test can either be a functional test, or also a code quality test.

Future work includes a more thorough analysis of the produced clusters with the Code Hunt dataset, but also with other datasets coming from learning platform. Automatic selection of the k parameter must also be explored and validation of the proposed analysis framework must also be done, evaluating the quality of the produced clusters.

5. REFERENCES

[1] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux. Code hunt: Experience with coding contests at scale. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, pages 398–407. ACM, May 2015.

[2] S. Combéfis, A. Bibal, and P. Van Roy. Recasting a traditional course into a mooc by means of a spoc. In *Proceedings of the European MOOCs Stakeholders Summit 2014 (EMOOCs 2014)*, pages 205–208, Feb. 2014.

[3] S. Combéfis and V. le Clément de Saint-Marcq. Teaching programming and algorithm design with pythia, a web-based learning platform. *Olympiads in Informatics*, 6:31–43, 2012.

[4] S. Combéfis and A. Paques. Pythia reloaded: an intelligent unit testing-based code grader for education. In *Proceedings of the 1st Int'l Code Hunt Workshop on Educational Software Engineering (CHESE 2015)*, pages 5–8. ACM, July 2015.

[5] S. Combéfis and J. Wautelet. Programming trainings and informatics teaching through online contests. *Olympiads in Informatics*, 8:21–34, 2014.

[6] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing*, 5(3), Sept. 2005.

[7] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction*, 22(2):7:1–7:35, Apr. 2015.

[8] E. L. Glassman, R. Singh, and R. C. Miller. Feature engineering for clustering student solutions. In *Proceedings of the 1st ACM Conference on Learning at Scale (L@S 2014)*, pages 171–172. ACM, Mar. 2014.

[9] J. Hage, P. Rademaker, and N. van Vugt. A comparison of plagiarism detection tools. Technical Report UU-CS-2010-015, Utrecht University, June 2010.

[10] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. 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 2010)*, pages 86–93. ACM, Oct. 2010.

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

[12] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, Inc., 2009.

[13] V. T. Martins, D. Fonte, P. R. Henriques, and D. da Cruz. Plagiarism detection: A tool survey and comparison. In *Proceedings of the 3rd Symposium on Languages, Applications and Technologies (SLATE 2014)*, pages 143–158, 2014.

[14] F. Murtagh. *Multidimensional Clustering Algorithms*. Physica-Verlag, 1985.

[15] K. A. Naudé, J. H. Greyling, and D. Vogts. Marking student programs using graph similarity. *Computers & Education*, 54(2):545–561, Feb. 2010.

[16] M. Novak. Review of source-code plagiarism detection in academia. In *Proceedings of the 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2016)*, pages 901–906, June 2016.

[17] V. Pieterse. Automated assessment of programming assignments. In *Proceedings of the 3rd Computer Science Education Research Conference (CSERC 2013)*, pages 45–56. ACM, Apr. 2013.

[18] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016–1038, Nov. 2002.

[19] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003)*, pages 76–85. ACM, June 2003.

[20] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, pages 15–26. ACM, June 2013.

[21] J. Sitthiworachart and M. Joy. Effective peer assessment for learning computer programming. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004)*, pages 122–126. ACM, June 2004.

[22] T. Staubitz, H. Klement, J. Renz, R. Teusner, and C. Meinel. Towards practical programming exercises and automated assessment in massive open online courses. In *Proceedings of the 2015 Annual IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE 2015)*, pages 23–30. IEEE, Dec. 2015.

[23] T. Wang, X. Su, Y. Wang, and P. Ma. Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2):99–107, Feb. 2007.

[24] Y. Wang, H. Li, Y. Feng, Y. Jiang, and Y. Liu. Assessment of programming language learning based on peer code review model: Implementation and experience report. *Computers & Education*, 59(2):412–422, Sept. 2014.