Toward Semi-Automatic Misconception Discovery Using Code Embeddings

Yang Shi
yshi26@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Krupal Shah
khshah2@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Wengran Wang
wwang33@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Samiha Marwan
samarwan@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Poorvaja Penmetsa
ppenmet@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Thomas W. Price
twprice@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

ABSTRACT

Understanding students’ misconceptions is important for effective teaching and assessment. However, discovering such misconceptions manually can be time-consuming and laborious. Automated misconception discovery can address these challenges by highlighting patterns in student data, which domain experts can then inspect to identify misconceptions. In this work, we present a novel method for the semi-automated discovery of problem-specific misconceptions from students’ program code in computing courses, using a state-of-the-art code classification model. We trained the model on a block-based programming dataset and used the learned embedding to cluster incorrect student submissions. We found these clusters correspond to specific misconceptions about the problem and would not have been easily discovered with existing approaches. We also discuss potential applications of our approach and how these misconceptions inform domain-specific insights into students’ learning processes.

CCS CONCEPTS

• Computing methodologies → Learning latent representations; • Applied computing → Education.

KEYWORDS

Neural Network, Code Analysis, Automatic Assessment, Learning Representation

1 INTRODUCTION

An accurate understanding of students’ misconceptions on a given problem is important for effective teaching and assessment of student knowledge. Researchers can improve the performance of student models by explicitly representing misconceptions [23]. Students can benefit from feedback on specific misconceptions, especially if those misconceptions are detected automatically (e.g. in [26]). Moreover, instructors can improve their lectures by explicitly addressing misconceptions, and better assess student knowledge by incorporating them into assignment rubrics. However, all of these applications first require researchers to discover what misconceptions exist in a given domain, or on a specific problem. For example, while instructors may already have rubrics to assess performance, these rubrics’ items may focus on qualities of the products created by students, rather than detecting specific misconceptions in student knowledge. Discovering such misconceptions is time-consuming and laborious, requiring experts to analyze traces of student work [20, 32] or interviews [5]. Such high cost of analysis makes it difficult to discover these problem-specific misconceptions.

Semi-automated misconception discovery can address these challenges by highlighting patterns in student data, which domain experts then inspect to identify misconceptions. However, existing discovery methods have focused on simple association rules [11, 16], rather than state-of-the-art deep learning methods. Deep learning methods have already been used to accurately assess student performance on complex tasks (e.g. essay writing [21]). In doing so, they also learn to represent students’ submissions as vectors, which makes it easier to visualize and cluster submissions and find meaningful patterns that correspond to students’ performance. This suggests that deep learning methods may be able to go beyond assessment and support the discovery of misconceptions.

In this work, we present a novel method for the semi-automated discovery of problem-specific misconceptions from students’ program code in computing courses, using a state-of-the-art, deep learning code classification model called code2vec [2]. Our approach trains code2vec to predict students’ success on a given problem, and then domain experts can use the visualizations produced using code2vec, along with students’ scores on a grading rubric, to quickly identify clusters of students who made similar mistakes on the problem. We applied this technique to one block-based programming problem from an introductory CS course for non-majors. We
find these clusters: 1) Correspond to specific misconceptions about the problem, which could be addressed with targeted feedback, and 2) Identify groups of students’ misconceptions that would not have been discovered from the rubric alone, or with standard code clustering approaches. Additionally, while code2vec has proven highly accurate at classifying code on large datasets, in this paper we investigate its effectiveness at the novel task of assessing student code on a much smaller dataset. We discuss potential applications of our approach (e.g., refining rubrics, informing instruction and feedback), and how these misconceptions inform domain-specific insights into students’ learning processes in a computing course. The primary contributions of this work are: (1) a novel semi-automated method for identifying problem-specific misconceptions (2) an evaluation of this method on one assignment in a introductory computer science (CS0) course, revealing insights into students’ specific learning processes and challenges, and (3) an evaluation of code2vec for automated code assessment, showing that it outperforms baseline models.

2 RELATED WORK

Misconceptions and Misconception Discovery: Misconceptions in computing education refer to an incorrect understanding of a concept or a set of concepts, which lead to making mistakes in writing or reading programs [34]. They are commonplace among novice learners but are many times invisible to experts and instructors [33] – a phenomenon explained in part by the “expert blind spot” [28]. Understanding misconceptions helps instructors to be more informed, and provides raw material for student modeling [22]. Prior work has explored ways to automatically detect students’ misconceptions [26], however, these detectors required experts to label a training set with the presence of misconceptions to train the model, which only works if we already know what misconceptions to look for. In CS education, misconceptions have been elicited through laborious manual analysis of students’ program code [29] or interviews [18]. In addition, manual construction of “bug libraries” can be a difficult and time consuming task for experts [4]. Nevertheless, such manual analysis, especially on a sequence of program snapshots, has been shown to provide more insightful information on students’ misconceptions. For example, Davies et al. manually analysed students’ programming trace data to discover misconceptions, and found that trace data revealed more knowledge gaps than what has been shown in their final submissions [6]. Other work focused on data-driven discovery of misconceptions. For example, Guzman et al. [16] used association rule mining to semi-automatically elicit misconceptions from multiple-choice questions in a CS practice test, and then allowed instructors to filter and verify meaningful misconceptions. However, such semi-automated detection of misconceptions also requires domain experts to analyze their rules to label them as constraints, and is only focused on closed-ended multiple choice questions (MCQs), rather than open-ended programming problems.

Automated Programming Code Assessment: Our approach builds on automated assessment technologies to detect misconceptions. Automated assessments help reduce instructor burden and enhance students’ learning. Much of the work on automated assessment using learning algorithms has been implemented in complex domains, such as open-ended learning environments [31] and essay grading [21]. However, most of the work in CS education domain has been limited to using test-cases for assessment (e.g. in [9]), rather than adapting these more advanced learning approaches. These test-driven assessment frameworks run a student’s code with hand-authored inputs, which requires additional instructor authoring effort and is difficult to apply to some problems (e.g. those with complex input, or graphical output). Some recent work has used machine learning algorithms to predict students’ learning outcomes by extracting more information for generalized and more complex problems [3, 25]. Building on this work, we leverage the code2vec model [2] to extract structural information for automated program assessment, which in turn forms the basis for our misconception discovery approach.

Assessment isn’t just about accuracy, it’s also about giving the student actionable feedback. Rubrics help us do this. Automated approaches have also been used to automatically extract rubrics. Clear rubrics provide interpretable justification of why students succeeded or failed an assessment [7]. Prior work has explored automatic discovery of rubrics. For example, Zhi et al. [35] and Diana et al. [7] automatically generated rubrics from students’ submissions by identifying common features of correct solutions. Dimopoulos et al. [8] created a Learning Management Systems plugin that enhanced rubrics with additional information about students’ interaction data (e.g. forum activity). However, none of these methods address how to improve rubrics by highlighting specific student misconceptions.

Code Analysis and Representation: Recent advances in deep learning models for code classification [2] suggest their potential for applications in education. One problem that these models solve is that neither code nor text representations can be directly translated into vector representations to be processed by machine learning algorithms. One way that natural language research solves this challenge is by using a trainable embedding matrix to represent different text contents [27]. Recent approaches represent the unstructured program code in a structured format that models can use as input. Among them, the most straightforward approach is to represent code as a sequence of one-hot encoders, which does not capture any of the structural information in code [1]. In CS education, Pech et al. [30] used a neural network to learn student code embeddings – an abstract vector representation of code – to identify which students would benefit from specific instructor feedback. However, this model considers only the output of code and ignores important structural information. They build their model based on the input and outputs of the code to extract the code embeddings, which does not leverage the important structural information for code itself. In this paper we focus on a particular approach, code2vec [2], which proposed expressive representations extracted from abstract syntax trees (ASTs), which greatly outperform prior work on code classification tasks. Recent work suggests that the embeddings produced by this approach may be particularly useful for finding patterns in student code [3]. In our work, we discover student misconceptions on a similar embedding matrix to represent code contents.

3 METHOD

The code2vec model learns a code embedding during training (highlighted in Figure 1), which extracts relevant structural information
We set out to answer the following research questions: RQ1: How accurately can a state-of-the-art code classification method assess students’ performance on a programming exercise? RQ2: How well does the code embedding from such a model capture meaningful similarities among students submissions, and do these reflect shared misconceptions?

4 EXPERIMENT

4.1 Dataset and Rubric

Our data come from an entry-level (CS0) computer science class for non-majors at a research university, spanning four consecutive semesters (Spring 2016 through Fall 2017). Students programmed in Snap, a block-based programming environment, designed for novices [14]. For this experiment, we chose to analyze one assignment from the course that required the use of loops, variables and procedures to draw a spiral shape, which proved particularly difficult for students (as it has a lowest average score among all assignments). We collected 207 submissions, which had each been graded using a rubric with 6 pass/fail items:

- **Procedure with 1 Parameter**: Create a procedure to draw a spiral shape with the given size.
- **Pen Down**: Use the “pen down” block in the procedure to start drawing.
- **Variable Init**: Initialize a variable and use it to control the length of each side of the shape.
- **Repeat Rotations**: Use a “repeat” block (i.e. a loop) to draw a spiral shape with the given size.
- **Forward + Turn**: Draw a square-like shape using the “forward” and “turn” blocks inside a loop.
- **Variable Increment**: Increment the side length variable after each iteration.

4.2 Creating and Validating Code2Vec Embeddings

Our goal is to use code2vec to create a meaningful embedding, which can be used to detect misconceptions. For consistency, we trained the code2vec model to predict whether a student got all rubric items correct (which 38% of students did). For a given training set, this yielded a single embedding, which we hypothesized would reflect learned code patterns indicating success on each rubric item. However, this embedding would only be meaningful if the model itself can accurately predict student success.

To evaluate the model’s performance, we compared code2vec to three baseline models: 1) a naive majority class baseline, 2) a support vector machine (SVM) (with a linear kernel, which outperformed Gaussian and Polynomial kernels), and 3) a fully connected neural network (NN). To convert student’s code into a vector of numeric features to use in the baseline models, we applied term frequency–inverse document frequency (TF-IDF) to each student’s code. This is similar to the Bag of Words approach used in prior learning analytics work [3] for code feature extraction. When evaluating all models’ performance, we split the Snap dataset into an 80% training dataset, and a 20% testing dataset. To ensure the robustness of our results in lieu of cross-validation, we re-sampled and re-trained the model 50 times and report the average performance.

**Hyperparameters**: In training both neural network models, we set the maximum training epochs as 10,000, with the patience of early stopping set to 400. We used a manual grid-search to select the learning rate (0.1 to 10\(^{-6}\)), number of linear layer dimensions (100, 200, 300) and batch size (32, 64, 128, full set), and found little difference across a hyper-parameters range. We thus selected the following parameters for both neural network models. The learning rate was set to 0.0002, linear layer dimensions were set to 100, and the batch size was set to the full set. The weights
We randomly selected a single code2vec model from the 50 runs and were learned during training using the Adam optimizer [19] for all these three models. On code2vec, we used 100-dimensional vectors for both nodes and paths. We applied the default values from the code2vec model, since changing these numbers within a range did not meaningfully change the results. We padded the number of paths to be the same length (100) over the student code dataset.

4.3 Clustering Incorrect Submissions to Feed Misconception Detection

We introduce the interpretations of embeddings to address RQ2 in this subsection, and include a case study in the next subsection. Table 2 shows the 4 densest clusters discovered for each rubric item (or as many as were discovered). Recall that the primary goal of our proposed method is to discover meaningful clusters (groups of students) who are demonstrating a similar misconception and would benefit from similar feedback. To verify this, one author, who is an instructor for the course, inspected each cluster and evaluated whether a shared misconception was present in the cluster, in which all (or nearly all) students failed the rubric item due to the same misconception. This process led to the discovery of 7 distinct misconceptions (shown in Table 2), which involved viewing 63 distinct submissions (about half of the incorrect submission). The author then created a short definition for each misconception.

Of the 17 clusters identified, only 4 had no consistent pattern or misconception (no label). This suggests that our approach frequently identified meaningful groups of students. To validate the original misconception labels, 2 other authors familiar with the class inspected the clusters and marked each submission as matching or not matching the misconception, resolving any disagreements. We report then number of “outliers” in each cluster which did not have the misconception. Many clusters had 0 outliers, and none had more than 1, further suggesting that clusters have internal consistency. One misconception (Move/Turn) showed up across almost every rubric item, often represented by similar or identical clusters of students. In the future, this redundancy could be avoided by excluding clusters that have high overlap with previously discovered clusters. To illustrate the types of misconceptions discovered by our approach, we describe 3 of the misconceptions discovered for one rubric item in detail below.

5.2 Embedding Interpretation

The first rubric item (referred to as R0) requires students to create a procedure with one parameter that holds the number of rotations of the spiral-square shape. This rubric was originally designed to test students’ ability to create procedures with parameters, which demonstrates an understanding of abstraction. While the rubric item was designed to assess a single student skill, we found that students failed the item for a wide variety of reasons, as illustrated by 3 distinct misconceptions:

The green cluster in Figure 2 contains 11 students who shared a misconception that iteration should be accomplished by duplicating code multiple times, (e.g. 5 pairs of “move” and “turn” blocks), rather than correctly repeating iterations with a loop. This misconception has little to do with R0 or with procedures (some students used a procedure; others did not), but it caused students to fail R0 since they could not correctly use the parameter without a loop. Unlike the other R0 clusters, these students needed basic feedback on a prerequisite programming concept (iteration), and would likely not benefit from feedback on procedures or parameters. These students
Table 2: Top-4 clusters Embedding Distance (ED) and Tree Edit Distance (TED), and the Interpretations of the clusters

| Rubric                        | Name and Description            | Size (# of students) | Outliers | TED    | ED     |
|-------------------------------|---------------------------------|----------------------|----------|--------|--------|
| R0 - Procedure with 1 Param.  | Fixed Shape                     | 7                    | 0        | 0.2912 | 0.1884 |
|                               | Large Move/Turn                 | 11                   | 1        | 1.9551 | 0.1657 |
|                               | Not testing a custom block       | 3                    | 0        | 0.2868 | 0.0813 |
|                               | Rotations not from Input         | 3                    | 0        | 0.4666 | 0.0788 |
| R1 - Initialize Variable      | Large Move/Turn                 | 10                   | 0        | 1.9599 | 0.1680 |
|                               | Medium Move/Turn                | 4                    | 0        | 0.5730 | 0.1254 |
|                               | Fail to initialize              | 3                    | 1        | 0.6041 | 0.0890 |
| R2 - Move and Turn            | Large Move/Turn                 | 6                    | 1        | 2.0339 | 0.1377 |
|                               | Medium Move/Turn                | 4                    | 1        | 0.4207 | 0.1515 |
| R3 - Repeat                   | Repeat sides not rotations      | 6                    | 0        | 0.4566 | 0.1357 |
|                               | Mixed incorrect repeat          | 11                   | 0        | 0.4719 | 0.2494 |
|                               | Mixed incorrect repeat          | 4                    | 0        | 0.9894 | 0.1342 |
|                               | Fixed shape                     | 4                    | 1        | 0.0891 | 0.0428 |
| R4 - Pen down                 | Mixed incorrect custom block    | 5                    | 0        | 0.5115 | 0.1436 |
|                               | Mixed incorrect custom block    | 4                    | 0        | 0.5449 | 0.1265 |
| R5 - Increment                | Large Move/turn                 | 9                    | 0        | 1.9781 | 0.1680 |
|                               | Medium Move/turn                | 5                    | 1        | 0.5947 | 0.1481 |

Figure 2: Code Embedding visualized clusters from the Procedure with 1 Input rubric item. We colored three clusters Rotations not from input (Brown), Fixed Shape (Orange) and Move/Turn (Green) for discussion in Subsection 5.3.

Figure 3: Pseudocode examples from student code dataset for the three clustered circled in Fig.2. Code related with misconceptions are highlighted with specific colors.

all failed other rubric items as well, and this misconception shows up consistently across rubric items.

The orange cluster in Figure 2 consists of 8 students who created a loop that repeats a fixed number of times, rather than using the procedure’s parameter to determine the number of rotations. This suggests a misconception about using loops with parameters. Unlike the previous cluster, students in this cluster knew how to create a loop, but they likely had a misconception that loops always iterated a fixed number of times, which was reflected in their inability to use a parameter or even a variable in the “repeat” block. Despite this similarity, there was also variety within the cluster: two of these students used a procedure without a parameter, while six did not create a procedure, and some used variables while others did not. Despite these syntactic differences, these submissions were positioned very close together in the embedding.

The brown cluster in Figure 2 consists of three students, where all of them used a local variable to determine the number of rotations, instead of the procedure’s parameter. In particular, two students created a procedure without parameters, then used an ask block to ask the user for an input and finally store it in a different created variable: “Rotations”. The third student did not create a procedure, and created a variable: “Rotations” and initialized it with a static value. Unlike the previous clusters, all students demonstrated an understanding that the size of the shape should be dynamic, using a variable in the repeat block. However, they shared a misconception about how a procedure parameter can be used to vary this value.
This could also be explained by a misunderstanding about the assignment requiring such a parameter, but in either case, students would benefit from targeted feedback on this misconception.

5.4 Are Clusters Distinct and Non-obvious?

Our proposed method is only useful if it detects clusters of student code that could not easily be otherwise identified. For example, if the clusters simply correspond to getting specific items in the rubric incorrect, there would be no need for a new approach to discover them. However, we found that the individual rubric scores varied widely within each cluster, without consistent patterns of which rubric items were gotten correct or incorrect (beyond the initial rubric item used to discover the cluster, which all students got incorrect).

Another possibility is that the clusters simply have similar ASTs, allowing existing code clustering methods to detect them. However, investigation of the code snapshots within each cluster revealed wide variety in the size and syntactic structure. To verify this, we calculated the average tree edit distance between each item in each cluster, and normalized this number by the average distance to the overall centroid. We compared this normalized cluster tree edit distance (TED) with the normalized embedding distance (ED), as shown in Table 2. We found the embedding distance is always much smaller than the tree edit distance, showing that the clustered submissions were relatively much closer together in the embedded space (compared to non-clustered submission), than in TED space, suggesting they would not be easily detected by traditional clustering methods. This shows the the embedding captures some aspects of code similarity which not apparent by simply comparing ASTs.

6 DISCUSSION

Implications for Research and Teaching: We envision a number of potential applications of our approach to help researchers and instructors to address students’ misconceptions. Feedback propagation: We found that students in the clusters we discovered would have benefited from targeted feedback that addressed their misconception. One application of our approach is in feedback propagation [17], where an instructor can write a single piece of feedback and send it to all students who might benefit. Moreover, our clusters would not have been discovered through traditional code clustering methods used in prior work (e.g. canonicalization [15]). Semi-automated feedback: Many of the misconceptions we discovered would be best addressed with formative feedback, as the student works. Since the patterns we discovered were relatively straightforward, one could imagine building a system to detect them automatically and offer appropriate immediate feedback early in a students’ work. In our investigation, we found that our clusters were not exhaustive, and did not include all students who shared the common misconception, just a representative subset. However, prior work suggests that, once a misconception is identified, models can be trained to accurately identify it automatically [26]. Rubric refinement: We found that a single rubric item can easily correspond to numerous, distinct challenges faced by students, as described in Section 5.3. Instructors and researchers may want to assess student knowledge in a more granular way, and our approach could help develop rubrics that capture the full range of students’ ability and knowledge. For example, it might be best to split the rubric item R0 into multiple items, or convert it from a binary pass/fail item to having scale values representing the various degrees of success.

Domain-Specific Implications for Computing Education: Another application of misconception discovery is to inform our understanding of a learning domain, and here we reflect on the implications of our findings for computing education. In this paper, we focused on R0, which assessed students’ ability to use a procedure parameter to vary the size of the shape they were drawing (determined by the number of iterations in a loop). While this rubric item was designed to assess a single concept (use of a parameter), our method discovered three clusters representing distinct barriers that students faced, requiring different remediation. The first cluster included students who had not figured out how to use a loop to draw the shape, suggesting the need for basic help on programming concepts needed in the assignment. The second cluster included students who were able to use a loop to draw the shape, but they did not understand that the size of the shape should vary; they used a static value instead. These students may need help understanding the assignment requirements, or which property of the shape should be parameterized. The third cluster of students did understand that the size of the shape should vary, but used a variable instead of the procedure parameter for this, suggesting their need to understand the use of parameters.

These results suggest implications for how to teach the lesson preceding this assignment, and how to teach procedure parameters in general. For our classroom, the instructor may want to do a more thorough review of loops and variables before the assignment (e.g. to help students in the first/green cluster). In general, our results suggest that students may struggle to use abstraction in the absence of a clear set of steps for creating a procedure with parameters. The instructor may want to teach students steps that address each of the barriers we identified, e.g.: 1) create a non-parameterized version of the procedure, which is the same every time, and make sure it works; 2) identify what aspect(s) of the procedure should be able to vary and create corresponding parameter(s); 3) replace the corresponding literal values in your code with the appropriate parameters. This aligns with a similar series of steps presented by the popular textbook on “How to Design Programs” to teach abstraction [13].

Limitations: This study is exploratory and has important limitations. First, we focused on one assignment from one course. This allowed us to discuss the misconceptions we discovered in depth, but as a result, it is not clear how well our results will generalize to other assignments. Second, our approach – and our evaluation – relied on the interpretations of experts to describe and label misconceptions, making the process semi-automatic. However, we argue this interpretation is a necessary part of making use of any misconceptions to verify their meaningfulness, and in our evaluation we used multiple raters to verify cluster meaning. Third, our approach yield some clusters that have no clear interpretation, and some duplicates, suggesting the need for future work to refine the process. Lastly, the discovered clusters are not exhaustive, meaning not all students who make a given misconception show up in the cluster. However, as discussed in Section 6, the clusters can still be useful to researchers and instructors, prompting changes to course structure that can benefit students not captured by the clusters.
7 CONCLUSION

In this paper, we presented a method for discovering student misconceptions semi-automatically using code assessment. We found that the code2vec model achieves a better semi-automated assessment performance than baseline models, showing that useful code structural information is learned in the model for the embeddings. We then examined if the embeddings could be used to discover misconceptions, and found that the embeddings generated clusters corresponding to meaningful and distinct misconceptions. From a case study on three clusters, we found that they are novel, consistent, and cannot be discovered by traditional code clustering methods, e.g. using tree edit distance. These results suggest that deep learning approaches have exciting potential to not only assess code, but help in the identification of learners’ misconceptions.

ACKNOWLEDGMENTS

This research was supported in part by the Amazon Web Services (AWS) Cloud Credits for Research program.

REFERENCES

[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In IJCAI ’16.
[2] Uri Alon, Mental Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. PopPL ’19 (2019).
[3] David Azcona, Piyush Arora, I-Han Hsiao, and Alan Smeaton. 2019. user2code2vec: Embeddings for profiling students based on distributional representations of source code. In LAK ’19.
[4] Paul T Baffes. 1994. Learning to Model Students: Using Theory Refinement to Detect Misconceptions. Technical Report. 1–34 pages.
[5] Ricardo Caceres, Steve Wolfman, Kellogg S Booth, and Rodolfo Azevedo. 2016. Developing a computer science concept inventory for introductory programming. In SIGCSE ’16.
[6] Randall Davies, Rob Nyland, John Chapman, and Gove Allen. 2015. Using Transaction-Level Data to Diagnose Knowledge Gaps and Misconceptions. In LAK ’15 (Poughkeepsie, New York). https://doi.org/10.1145/2733576.2733620
[7] Nicholas Diana, Michael Eagle, John Stamper, Shuchi Grover, Marie Bienkowski, and Satabdi Basu. 2018. Data-driven generation of rubric criteria from an educational programming environment. In LAK ’18. 16–20.
[8] Ioannis Dimopoulos, Ourania Petropoulou, and Symeon Retalis. 2013. Assessing students’ performance using the learning analytics enriched rubrics. In LAK ’13. 195–199.
[9] Stephen H Edwards and Krishnan Panamalai Murali. 2017. CodeWorkout: short programming exercises with built-in data collection. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education. 188–193.
[10] Mohammed TH Elhotta and Wesam M Ashour. 2013. A dynamic method for discovering density varied clusters. International Journal of Signal Processing, Image Processing and Pattern Recognition 6, 1 (2013).
[11] Myse Elmadani, Moffat Mathews, and Antonija Mitrovic. 2012. Data-driven misconception discovery in constraint-based intelligent tutoring systems. In Proceedings of the 20th International Conference on Computers in Education. http://ir.canterbury.ac.nz/handle/10092/7399
[12] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. AAAI Press, 226–231.
[13] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. How to design programs: an introduction to programming and computing. MIT Press.
[14] Dan Garcia, Brian Harvey, and Tiffany Barnes. 2015. The Beauty and Joy of Computing. ACM Inroads 6, 4 (2015), 71–79.
[15] Elena I Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. ACM Transactions on Computer-Human Interaction (TOCHI) 22, 2 (2015), 1–35.
[16] Eduardo Guzmán, Ricardo Conejo, and Jaime Galván. 2010. A data-driven technique for misconception elicitation. UMAP’10 6075 LNCs (2010), 243–254.
[17] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Fignseredo, Loris D’Antoni, and Bjorn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In Proceedings of the ACM Conference on Learning @ Scale. 89–98. https://doi.org/10.1145/3051457.3051467