SelfCode: An Annotated Corpus and a Model for Automated Assessment of Self-explanation during Source Code Comprehension

Jeevan Chapagain[1], Zak Risha[2], Rabin Banjade[1], Priti Oli[1], Lasang Jimba Tamang[1], Peter Brusilovsky[2], Vasile Rus [1]

[1] Department of Computer Science, Institute of Intelligent System, University of Memphis, Memphis, TN, USA
[2] School of Computing and Information, University of Pittsburgh, Pittsburgh, PA, USA
{jchpgain,rbanjade1,poli,ljtamang,vrus}@memphis.edu, {zjr9,peterb}@pitt.edu

Abstract

The ability to automatically assess learners' activities is the key to user modeling and personalization in adaptive educational systems. The work presented in this paper opens an opportunity to expand the scope of automated assessment from traditional programming problems to code comprehension tasks where students are requested to explain the critical steps of a program. The ability to automatically assess these self-explanations offers a unique opportunity to understand the current state of student knowledge, recognize possible misconceptions, and provide feedback. Annotated datasets are needed to train Artificial Intelligence/Machine Learning approaches for the automated assessment of student explanations. To answer this need, we present a novel corpus called SelfCode which consists of 1,770 sentence pairs of student and expert self-explanations of Java code examples, along with semantic similarity judgments provided by experts. We also present a baseline automated assessment model that relies on textual features. The corpus is available at GitHub repository [1].

Introduction

Assessment is a central task in education in general and adaptive education technologies in particular (Chi, Siler, and Jeong 2004). Assessing students' knowledge states (and other states, e.g., affect state) or building reliable learner models is key to personalized instruction because it enables macro- and micro-adaptation of instruction. Macro-adaptation is the selection of appropriate instructional tasks (also called task-level adaptation) given a student's state. Micro-adaptation (or within-task adaptation) means monitoring and providing appropriate scaffolding within a task, such as a problem to solve or code to understand. This paper expands the range of student activities an adaptive educational system can assess, from traditional programming problems to code comprehension tasks. Its broader focus is on building adaptive educational systems to help students improve their code comprehension skills.

Source code comprehension means identifying the functional pieces of a computer program (code) and how they relate to one another to offer the holistic, higher-level functionality of the code, i.e., how they serve the overall aim of the code and how the functional parts are brought about utilizing computational concepts and methods. Code comprehension is essential for both learners and professionals. For instance, learners trying to learn computer programming spend significant time looking at code examples from their instructor or a textbook. Similarly, understanding code is the most time-consuming process in software maintenance, responsible for 70% of a software product's overall life-cycle cost (Rugaber 2000) as source code understanding is essential when a programmer maintains, reuses, migrates, re-engineers, or upgrades software systems (O'Brien 2003).

Given the central role of code comprehension for both learners and professional programmers, it is important to help learners acquire advanced code comprehension skills. Indeed, providing support to help students improve their source code comprehension skills could have long-term benefits for their academic progress and future professional careers. Our efforts to develop a code comprehension intelligent tutoring system (ITS) that monitors, models, and scaffolds learners' code comprehension processes rely on text comprehension strategies such as self-explanations. The success of self-explanation for text comprehension and learning can be linked to its constructive aspect, e.g., it activates several cognitive processes such as generating inferences to fill in missing information and integrating new information with prior knowledge, monitoring and repairing faulty knowledge, and its meaningfulness for the learner, i.e., self-explanations are self-directed and self-generated making the learning and target knowledge more personally meaningful, in contrast to explaining the target content to others. Self-explaining has demonstrated a positive impact on student learning in a variety of fields, including physics (Conati and VanLehn 2000), math (Aleven and Koedinger 2002), and programming (Bielaczyc, Pirolli, and Brown 1995; Rus et al. 2021).

Self-explanation prompts can emphasize various aspects of self-explanations resulting in justification-based self-explanation prompts (Chen et al. 2020) or meta-cognitive self-explanation prompts (Chi et al. 1994). It has been observed that professional programmers are prompted to self-explain and follow one of the three types of code comprehension strategies – top-down, bottom-up, or opportunistic – however, more studies are necessary to understand how those strategies may work for beginners and which one is

[1] https://github.com/jeevanchaps/SelfCode

the best for whom under what circumstances. It may be the case that a combination of those strategies may work for beginners, e.g., a top-down strategy implemented as support from the tutor/(human) instructor in which the major functional blocks are identified while the learner reads the lines of the code in those blocks and infers the functionality of each of the blocks (a bottom-up strategy within the block) based on which the overall goal of the code is then inferred.

The work presented here is a step forward in identifying the type of self-explanations that work best for learners. In this step, we focus on self-explanations that are guided by a bottom-up strategy mainly as we ask learners to read and explain each line of code as opposed to, for instance, block-level explanations, which are more appropriate for a top-down comprehension strategy in which the reader first identifies the major functional blocks of the code before analyzing each line of code in detail. It is important to note that we plan to explore the role of top-down strategies in the future, but that is beyond the scope of the effort presented in this paper.

More specifically, the work presented here focuses on developing a corpus called SelfCode (*self*-explanation of *code*), which contains a wide variety of "native" (i.e., non-expert) code self-explanations paired with expert explanations and judgments of semantic similarity between native and expert self-explanations of the same code lines. We applied a crowdsourcing approach to collect diverse explanations from users with different backgrounds and skills. Expert explanations were acquired from a collection of expert-annotated examples from a catalog of interactive learning content (Hicks et al. 2020). Examples of expert and crowdsourced explanations of code lines are shown in Table 1. Figure 1 shows the interface used for the crowdsourced collection of line-by-line self-explanations.

We then manually annotated each expert and crowdsourced self-explanation sentence pairs using a 1-5 scale semantic similarity scale, where 1 is not similar (incorrect and/or irrelevant) to the expert explanation, whereas 5 means semantically equivalent. The resulting SelfCode dataset contains 1,770 sentence pairs of crowdsourced and expert self-explanations.

The role of expert explanations in our dataset is to serve as a benchmark (ground truth) for automated methods that assess the correctness of students' self-explanations. Such methods assess correctness by computing the semantic similarity between the student self-explanations and the expert self-explanations. Using the semantic similarity approach, if a student's self-explanation is semantically similar to the corresponding expert explanation, then the student's self-explanation is considered to have the same correctness level as the expert explanation. Our "gold standard" similarity scores allow our dataset to be used for training and testing this type of automated correctness-assessment method.

This paper discusses the value of the SelfCode-type datasets and reviews important steps of collecting and annotating student self-explanations of code examples. At the end of the paper, we demonstrate the application of this dataset to training and testing a number of similarity assessment approaches, which could serve as baselines for future research in this area.

Example code line	**Expert explanation**	**Crowd sourced explanation**
Point1 point = new Point1()	The variable point holds a reference to a Point1 object.	Create a new Point1.
System.out.println ("The integer is positive")	This statement prints that the integer is positive.	Print that the number is positive if it is greater than 0.
String fullname = "John Smith"	We define a string variable to hold the name.	Sets the fullName string to John Smith.
translate(11,6)	This line invokes the method translate of the point.	moves the X,Y points by adding (11,6) point.

Table 1: Crowdsourced line-by-line self-explanation vs. Expert Explanation with the corresponding line of JAVA code shown in Figure 1.

The Potential of Freely Generated Self-Explanations

Freely generated student self-explanations enable students to freely articulate their thinking as opposed to, for instance, multiple-choice questions where students choose an answer from a set of given answers (only one of which is typically correct) without providing any explanations. One major risk of using multiple-choice questions is that students may pick the correct answer for the wrong reasons without the assessor knowing it. While one can argue that multiple-choice questions can be augmented with requests for explanations, the student must explain their answer choice. However, this brings us back to the problem of assessing open-ended responses. It should be noted that there is a subtle difference between multiple-choice-with-explanation questions and traditional open-ended questions in that the former gives students more information to work with, including the correct answer. Some students will be able to recognize the correct answer and, in retrospect, generate an explanation. In contrast, in a pure open-ended question assessment scenario, students are supposed to both generate the answer and a solid explanation without any extra hints in the form of a set of potential answer choices.

Assessing students' open-ended responses is a complex problem. While general solutions are preferred, state-of-the-art solutions are specialized for specific instances of the problem that account for the context in which the responses are generated, e.g., short open-ended responses in the middle of a tutoring session in a particular domain such as conceptual Physics. Depending on the wider context in which they are generated, we categorize open-ended responses into

three major classes: (1) open-ended questions used in assessment instruments that typically require a paragraph-length short essay answer, (2) student utterances in tutorial dialogues in response to tutor prompts, which usually involve a very short, highly contextualized response (such as "equal" as a response to the computer tutor's question about how the magnitude of two physics forces compare), and (3) assessing longer essays in response to SAT-like prompts (SAT – Scholastic Aptitude/Assessment Test, a standardized test widely used for college admission in the United States and taken by millions of high school students each year). While automatically assessing students' free responses has been explored extensively before, it has been less studied in the context of code comprehension, the focus of this work.

In this work, our goal is to automatically assess student-generated free explanations of code when prompted to explain line-by-line those code examples (as opposed to, for instance, asking them to identify and explain the functionality of the major blocks of code).

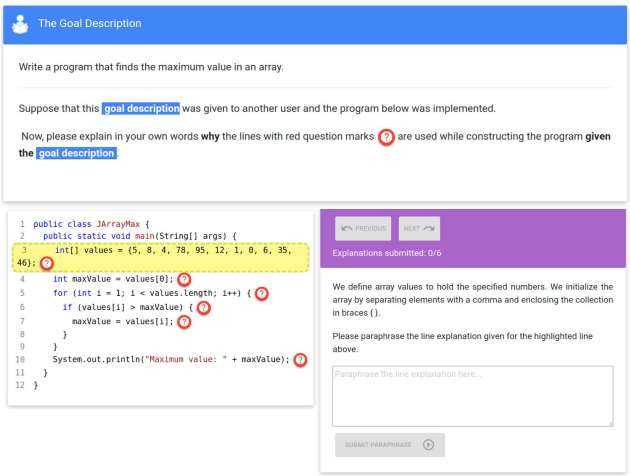


Figure 1: An interface for collecting line-by-line self-explanations of a worked example - a model solution of a programming problem.

Related Work

The development of SelfCode was guided by a number of theories and frameworks, such as the self-explanation theory (Chi 2000; O'Brien 2003), as well as breakthroughs in code comprehension and text comprehension research (Brooks 1983; Graesser, Singer, and Trabasso 1994; Good 1999; Pennington 1987) and recent advances in auto-assessment of freely-generated student answers (Banjade et al. 2015).

The importance of building accurate mental models during learning tasks has been well established for decades in domains like science (De Jong and Ferguson-Hessler 1991) as well as in CS education (Soloway and Ehrlich 1984; Pennington 1987; Ramalingam, LaBelle, and Wiedenbeck 2004). Prior CS education research has documented the many difficulties novice programmers face while learning to program, such as constructing accurate mental models during key learning activities (Ramalingam, LaBelle, and Wiedenbeck 2004). The challenge with constructing accurate mental models is not surprising given that constructing mental representations is considered a higher-level comprehension skill, typically engendering a high cognitive load (Kintsch and Walter Kintsch 1998; Graesser and McNamara 2011).

Prompting for self-explanations is a helpful strategy that has shown promising results with respect to helping students become better comprehenders and learners. As noted previously, self-explanations are helpful for learning because they involve various cognitive processes, such as drawing inferences to fill in knowledge gaps, fusing new knowledge with prior knowledge, and observing and correcting incorrect knowledge (Roy and Chi 2005). It has been discovered that written self-explanation is more suited for students who struggle with challenging reading activities, such as reading scientific texts, which have a higher cognitive load. Research conducted on undergraduates (Rezel 2003) and high school students (Alhassan 2017) showed students who self-explained were better at program construction than those who did not. Another conclusive effect of self-explanation can be seen in the series of studies focusing on LISP programming (Recker and Pirolli 1990; Pirolli and Recker 1994). The positive effect of self-explanations has been demonstrated in learning various languages JavaScript (Kwon and Jonassen 2011) and assembly language (Hung 2012).

If self-explanation is to be used in an adaptive instructional system, students' self-explanation must be automatically assessed in order to assess their comprehension level and provide adequate feedback. With respect to building automated methods for assessing self-explanations, related resources such as annotated datasets to train-test automated methods using supervised methods are needed. There has been plenty of work in this area (Rus, Banjade, and Lintean 2014) but not in the context of code comprehension, which can be more challenging as students' free explanations mix natural language with code/programming language constructs and program-specific references such as variable names or task-specific lingo such as referring to Bingo game terms in the case of a code example that generates Bingo boards. Hence, the need to develop an automated method for assessing students' self-explanations during code comprehension and the related needed resources such as the novel dataset and some baseline auto-assessment methods presented here. To the best of our knowledge, no such annotated dataset exists for the domain of intro-to-programming and code comprehension. The most relevant related works are highlighted below.

A comprehensive overview of paraphrase identification corpora, including datasets for assessing student answers, is provided in (Rus, Banjade, and Lintean 2014). Paraphrase identification is a closely related task to computing the semantic similarity for assessing student-free responses. For instance, the paraphrasing task can be framed as a binary, qualitative task in which the outcome is a binary decision about whether two short texts paraphrase each other, i.e., convey the same meaning differently.

SimLex-999 (Hill, Reichart, and Korhonen 2015) is an important dataset for assessing distributive semantic models that measure similarity rather than relatedness. The annotation protocol they used assigns a low ranking to pairs of connected items that are not actually similar.

Banjade and colleagues (Banjade et al. 2016) developed the DT-Grade corpus comprising short generated responses collected from tutorial conversations between students and an ITS for the domain of Newtonian Physics. They annotated the student responses for correctness given the larger context (not only the student self-explanation and the expert explanation in isolation). Also, they judged the usefulness of contextual information to correctly assess the student response in order to overcome some limitations of prior efforts in which student's short responses and the corresponding expert explanation were provided as input ignoring the larger context such as the Physics problem the student is tasked to solve or the prior dialogue history between the learner and the tutor.

Mohler and Mihalcea (Mohler and Mihalcea 2009) released a collection of short student responses and grades for a computer science course to assess student responses based on textual similarity. However, their dataset was not in the context of code comprehension.

Data Collection and Annotation

We created the SelfCode dataset starting with a subset of the open-source collection of expert-annotated Java code examples available from a catalog of interactive learning content (Hicks et al. 2020). In total, 10 Java code examples were selected to form the core of our dataset. These examples were extended with line-by-line self-explanations, and then we added semantic similarity expert judgments.

Crowdsourced Data Collection

Amazon Mechanical Turk (MTurk) was selected as a venue for collecting code self-explanations since this crowdsourcing platform provides access to a wide range of workers, including workers with programming experience. To collect the self-explanations, we developed a dedicated interface that prompts crowd workers to explain each line of code in our selected worked examples (Fig. 1). To ensure the quality of collected data, we focused our recruitment on responsible workers with demonstrated Java programming knowledge. The code examples to be explained through this interface were offered as Human Intelligence Tasks (HITs) to workers from the US and Canada with a track record of at least 100 HITs approved and a 97% approval rate. Participants were informed they would be shown a Java program for which they should have to provide self-explanations for each line of code. Crowd workers had to answer three multiple-choice program construction questions by selecting the correct missing line to qualify for this work. Participants needed to answer 2 out of 3 tasks correctly to obtain qualification — multiple attempts were not permitted.

For our first round of explanations, crowd workers were provided a Java program and a goal description, framed as high-level instructions given to another user who implemented the program. Users were then prompted to "Please explain in your own words why the lines with red question marks are used while constructing the program given the goal description." A total of 10 Java programs were used across 5 HITs. Workers were compensated monetarily.

After reviewing the initial set of explanations, we attempted to increase the diversity of types of explanations and ensure that crowd workers were more focused on explaining the code. Our second round attempted to elicit two types of explanations: (1) *why* a line was necessary given the goal of the program, i.e., identifying a subgoal the line achieves, and (2) *how* it achieves the particular subgoal, i.e., explanations that describe the behavior of a line. We tried to guide the participants by providing one type of explanation; in this case, *why* a line was included in the program. Participants were then prompted "How does this line achieve the above goal? What action does it perform?" We expected that by providing one form of explanation as an example, we would help the crowd workers differentiate between different aspects of the code. The second round used 5 Java programs spread across 3 HITs, less than the first round, to elicit focused responses. Workers were compensated monetarily.

Throughout all HITs, we had a total of 30 unique crowd workers. The workers of the first round were not eligible to participate in the second round. Of these workers, 90% correctly answered all three qualification questions, suggesting Java competency. In our first round, we collected a total of 574 line explanations for the 10 Java programs. In the second round, we collected a total of 221 line explanations for 5 Java programs.

Annotation

Once the data was collected, we annotated it with expert judgments of self-explanation correctness. Since we had available expert explanations, the annotation aimed to judge whether the crowd workers provided self-explanation for a given line of code and whether the expert-provided explanation was semantically equivalent (or not).

There is another important methodological step we must mention. Both participants' self-explanations and the expert explanations for each line of code may contain more than one sentence. For annotation purposes, we decided to pair sentences and, therefore, use a sentence as the unit of analysis, as sentences are the basic language unit expressing a full idea. There are two other important reasons to work at the sentence level: (1) prior semantic similarity corpora focused on sentence-level similarity, and we wanted to maintain some level of compatibility to prior efforts for comparison purposes, and (2) many previously developed state-of-the-art automated methods for semantic similarity work at the sentence level (they can be expanded to paragraph level in various ways as explained below). Therefore, crowdsourced and expert explanations were broken down into individual sentences. The sentences were paired with each other, each pair consisting of a participant self-explanation sentence and an expert-generated explanation sentence, resulting in a dataset of 1,770 sentence pairs.

The annotation was performed by 6 Ph.D. students proficient in computer programming. They were first trained on the annotation guidelines before doing any annotations. The

goal of the annotation was twofold: (1) separate why/goal-oriented explanations from what/behavior-oriented explanations and (2) judge the semantic similarity between the crowdsourced explanations and the expert explanations.

The annotation protocol followed the main steps presented below:

- For each sentence in an expert explanation, a judgment was first made whether the sentence is related to the overall goal of the corresponding code example (why the line is needed given this goal) or describes the behavior (how) of the code.
- For each sentence in the crowd-sourced self-explanation, a judgment was made with respect to their semantic similarity to the corresponding expert sentence using the following rating scale: 5 - semantically equivalent, 4 - almost semantically equivalent, meaning the rater has high confidence that the student explanation has high coverage of the concepts in the expert explanation, 3 - medium coverage of the concepts mentioned in the expert explanation, 2 - low coverage of the concepts mentioned in the expert explanation, and 1 - incorrect/irrelevant. For the latter category, incorrect/irrelevant annotators were supposed to make a note if a major misconception was present in the collected self-explanation.

Three students annotated each sentence pair, with three students rating the first half of the data and another three students annotating the second half. The annotation was completed in two stages. In the first round, each annotator provided judgments for the first 100 sentence pairs; disagreements in the ratings were discussed to resolve them and reach a common understanding of the annotation protocol. Once the rating protocol was well understood and consistently applied by all for the first 100 pairs, a second round of the annotation was completed for the rest of the instances. After the second round, a disagreement mitigation step followed with the goal of bringing the scores within at most 1 point difference among annotators. Due to slight differences in interpreting the sentences, we viewed a difference of zero or one as "agreement". With a couple of exceptions, each sentence pair received a similarity rating that differed by no more than 1 point among the three annotators (except in just a few situations). We then took the average of annotators' ratings to generate a single rating score.

Using Fleiss Kappa (Fleiss 1971), inter-rater agreement was computed, resulting in an inter-rater agreement kappa of 0.33 for the first round and 0.99 for the second round, indicating a fair and great agreement, respectively, among the annotators.

Dataset

Table 2 offers a descriptive statistics summary of the resulting dataset of 1,770 annotated sentence pairs. 17.95% of sentence pairs contained a rating score of 4 or 5, while 58.50% of sentence pairs were incorrect (score 1) or had low coverage of the concepts (score 2).

Annotation Label	No. of sentence pairs
1	529 (29.88%)
2	507 (28.62%)
3	419 (23.65%)
4	253 (14.45%)
5	62 (3.50%)

Table 2: Summary of SelfCode dataset

Baseline Models: Experiments and Results

Our main goal was to create a dataset to foster the development of state-of-the-art natural language processing methods for automatically assessing student explanations during source code comprehension tasks.

To this end, we experimented with developing baseline automated methods for assessing the student explanations in the SelfCode dataset. To build those methods, we extracted textual features from each sentence pair and used those features in a machine learning model trained using a number of supervised machine learning algorithms.

The textual features of the model are the total word count difference between the crowd-sourced self-explanation and the expert explanation, the number of overlapping words as a way to capture content/semantic overlap, and the number of overlapping word bigrams as a way to capture word order (syntax) information and a semantic similarity score obtained using SentenceBERT (Reimers and Gurevych 2019) embeddings. The features were then used with four different machine learning algorithms: Logistic Regression (LR), Decision Tree (DT), Support Vector Machine (SVM), and Naive Bayes (NB) using different feature combinations.

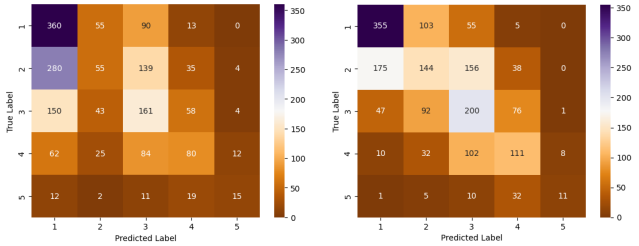
Models	Precision	Recall	F1-score	Accuracy
LR	0.30	0.27	0.25	36.91%
DT	0.30	0.30	0.29	33.24%
SVM	0.18	0.21	0.15	30.69%
NB	**0.36**	**0.35**	**0.32**	**37.93%**

Table 3: Performance of the models with textual features (M1)

Models	Precision	Recall	F1-score	Accuracy
LR	**0.37**	**0.37**	**0.36**	**47.31%**
DT	0.32	0.33	0.32	37.25%
SVM	0.18	0.21	0.15	30.92%
NB	0.43	0.41	0.40	46.40%

Table 4: Performance of the models with textual features and semantic similarity score from SentenceBERT (M2)

To accurately assess the performance of our models with the limited dataset of 1770 sentence pairs, we used a 10-fold stratified cross-validation technique. We also used Synthetic Minority Oversampling Technique (SMOTE) to balance the data across all the classes, but the result showed overfitting due to oversampling, so we discarded the oversampling tech-



(a) (b)

Figure 2: Confusion matrix for M1 and M2 respectively

nique. Table 3 & Table 4 shows the result of the 10-fold stratified cross-validation experiment using textual features (M1) and using textual as well as the embedding feature (M2), respectively.

We then took the best-performing models and explored their performance for each of the 1-5 ratings. Figure 2a and Figure 2b show the plot for the confusion matrix for best-performing models in M1 and M2. The confusion matrix in the figure indicates that models M1 and M2 generally make correct predictions for their respective classes, with relatively high values in the diagonal elements. However, they struggle to distinguish the correct class for instances with a score of 5, potentially due to either the model's failure to capture sufficient contextual information for the class with a score of 5 in comparison to the other scores or it was adversely affected by the issue of class imbalance.

Crowd Sourced Explanation	**Standard Explanation**	**M1**		**M2**	
		A	P	A	P
It declares an Array of integers.	We initialize the array by separating elements with a comma and enclosing the collection in braces	3	1	3	3
Ask the user to input an integer for seconds.	We prompt the user to enter the seconds.	5	2	5	4
set the maxValue to the value at position 0 in the value array.	We need variable maxValue to store the maximum value of the array.	4	2	4	3
It closes this scanner.	We close the scanner as we do not want to process any input from the user in the rest of the program.	2	4	2	1

Table 5: Comparison of models **M1** and **M2** where the prediction of the models varied. Columns **A** and **P** represent the Actual and Predicted scores respectively.

Error Analysis

Table 5 shows a number of self-explanations for which the predicted semantic similarity score differs from the actual gold score obtained from the human annotators. It should be noted that the two models, M1 and M2, are pretty simple, i.e., they are baseline models, and therefore, they are not expected to perform very well. Nevertheless, this error analysis for those baseline models can reveal and inform the future development of more advanced models based on the SelfCode dataset. From the examples in the table, we note the following. First, the expert explanations seem to overuse the personal pronoun 'we,' whereas crowd workers do not self-explain the code in this style. One fix is to train students to use the same style or discard or ignore this pronoun when computing the semantic similarity automatically. Second, the expert explanations are much longer, which suggests a two-stage assessment approach may be useful, i.e., in the first stage, the crowd workers' explanation is first assessed to be correct, and in the second stage, whether it is complete. This is something to explore in future work.

Conclusion

We presented the development of a corpus called SelfCode which consists of crowdsourced line-by-line Java code self-explanations, expert explanations for the same lines, and expert-provided similarity scores. The dataset was designed to assist the development of supervised machine-learning methods for automated assessment of the correctness and similarity of student explanations of code comprehension in introductory programming. We hope that SelfCode will stimulate the development of state-of-the-art assessment methods for code comprehension and eventually develop fully-fledged intelligent tutoring systems that help students become better code readers. Our future work includes: a) extending the single expert explanations to multiple sentences, which helps to provide adequate explanations compared to a single sentence. b) check the quality of the explanations provided by crowd workers, which can also be added as alternate explanations that help make the dataset richer.

References

Aleven, V. A., and Koedinger, K. R. 2002. An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. *Cognitive Science* 26(2):147–179.

Alhassan, R. 2017. The effect of employing self-explanation strategy with worked examples on acquiring computer programing skills. *Journal of Education and Practice* 8(6):186–196.

Banjade, R.; Niraula, N. B.; Maharjan, N.; Rus, V.; Stefanescu, D.; Lintean, M.; and Gautam, D. 2015. NeRoSim: A system for measuring and interpreting semantic textual similarity. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, 164–171. Denver, Colorado: Association for Computational Linguistics.

Banjade, R.; Maharjan, N.; Niraula, N. B.; Gautam, D.; Samei, B.; and Rus, V. 2016. Evaluation dataset (dt-grade) and word weighting approach towards constructed short answers assessment in tutorial dialogue context. In *Proceedings of the 11th Workshop on Innovative Use of NLP for Building Educational Applications*, 182–187.

Bielaczyc, K.; Pirolli, P. L.; and Brown, A. L. 1995. Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem-solving. *Cognition and Instruction* 13(2):221–252.

Brooks, R. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18(6):543–554.

Chen, B.; Azad, S.; Haldar, R.; West, M.; and Zilles, C. 2020. A validated scoring rubric for explain-in-plain-english questions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 563–569.

Chi, M. T.; De Leeuw, N.; Chiu, M.-H.; and LaVancher, C. 1994. Eliciting self-explanations improves understanding. *Cognitive Science* 18(3):439–477.

Chi, M. T.; Siler, S. A.; and Jeong, H. 2004. Can tutors monitor students' understanding accurately? *Cognition and Instruction* 22(3):363–387.

Chi, M. 2000. Self-explaining: The dual processes of generating inference and repairing mental models. In Glaser, R., ed., *Advances in Instructional Psychology: Educational Design and Cognitive Science. Vol. 5*. Mahwah, NJ: Lawrence Erlbaum Associates. 161–238.

Conati, C., and VanLehn, K. 2000. Further results from the evaluation of an intelligent computer tutor to coach self-explanation. In *International Conference on Intelligent Tutoring Systems*, 304–313. Springer.

De Jong, T., and Ferguson-Hessler, M. G. 1991. Knowledge of problem situations in physics: A comparison of good and poor novice problem solvers. *Learning and Instruction* 1(4):289–302.

Fleiss, J. L. 1971. Measuring nominal scale agreement among many raters. *Psychological Bulletin* 76(5):378.

Good, J. 1999. *Programming Paradigms, Information Types, and Graphical Representations: Empirical Investigations of Novice Program Comprehension*. Ph.D. Dissertation, University of Edinburgh.

Graesser, A. C., and McNamara, D. S. 2011. Computational analyses of multilevel discourse comprehension. *Topics in Cognitive Science* 3(2):371–398.

Graesser, A. C.; Singer, M.; and Trabasso, T. 1994. Constructing inferences during narrative text comprehension. *Psychological Review* 101(3):371.

Hicks, A.; Akhuseyinoglu, K.; Shaffer, C.; and Brusilovsky, P. 2020. Live catalog of smart learning objects for computer science education. In *Sixth SPLICE Workshop "Building an Infrastructure for Computer Science Education Research and Practice at Scale" at ACM Learning at Scale 2020*.

Hill, F.; Reichart, R.; and Korhonen, A. 2015. SimLex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics* 41(4):665–695.

Hung, Y.-C. 2012. Combining self-explaining with computer architecture diagrams to enhance the learning of assembly language programming. *IEEE Transactions on Education* 55(4):546–551.

Kintsch, W., and Walter Kintsch, C. 1998. *Comprehension: A paradigm for cognition*. Cambridge University Press.

Kwon, K., and Jonassen, D. H. 2011. The influence of reflective self-explanations on problem-solving performance. *Journal of Educational Computing Research* 44(3):247–263.

Mohler, M., and Mihalcea, R. 2009. Text-to-text semantic similarity for automatic short answer grading. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, 567–575.

O'Brien, M. P. 2003. Software comprehension–a review & research direction. *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report*.

Pennington, N. 1987. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop, 1987*, 100–113.

Pirolli, P., and Recker, M. 1994. Learning strategies and transfer in the domain of programming. *Cognition and Instruction* 12(3):235–275.

Ramalingam, V.; LaBelle, D.; and Wiedenbeck, S. 2004. Self-efficacy and mental models in learning to program. In *Proceedings of the 9th SIGCSE Conference on Innovation and Technology in Computer Science Education*, 171–175.

Recker, M. M., and Pirolli, P. 1990. A model of self-explanation strategies of instructional text and examples in the acquisition of programming skills. In *Annual Meeting of American Educational Research Association*. ERIC.

Reimers, N., and Gurevych, I. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Rezel, E. 2003. *The effect of training subjects in self-explanation strategies on problem-solving success in computer programming*. Ph.D. Dissertation, Marquette University.

Roy, M., and Chi, M. T. 2005. The self-explanation principle in multimedia learning. *The Cambridge Handbook of Multimedia Learning* 271–286.

Rugaber, S. 2000. The use of domain knowledge in program understanding. *Annals of Software Engineering* 9(1):143–192.

Rus, V.; Banjade, R.; and Lintean, M. 2014. On paraphrase identification corpora. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, 2422–2429.

Rus, V.; Akhuseyinoglu, K.; Chapagain, J.; Tamang, L.; and Brusilovsky, P. 2021. Prompting for free self-explanations promotes better code comprehension. In *5th Educational Data Mining in CS Education Workshop at EDM2021*.

Soloway, E., and Ehrlich, K. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* (5):595–609.