SelfCode 2.0: An Annotated Corpus of Student and Expert Line-by-Line Explanations of Code for Automated Assessment

Jeevan Chapagain^1, Arun Balajiee Lekshmi Narayanan^2, Kamil Akhuseyinoglu^2
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,vrus}@memphis.edu, {arl122,kaa108,peterb}@pitt.edu

Abstract

Assessing student responses is a critical task in adaptive educational systems. More specifically, automatically evaluating students' self-explanations contributes to understanding their knowledge state which is needed for personalized instruction, the crux of adaptive educational systems. To facilitate the development of Artificial Intelligence (AI) and Machine Learning models for automated assessment of learners' self-explanations, annotated datasets are essential. In response to this need, we developed the SelfCode2.0 corpus, which consists of 3,019 pairs of student and expert explanations of Java code snippets, each annotated with semantic similarity, correctness, and completeness scores provided by experts. Alongside the dataset, we also provide performance results obtained with several baseline models based on TF-IDF and Sentence-BERT vectorial representations. This work aims to enhance the effectiveness of automated assessment tools in programming education and contribute to a better understanding and supporting student learning of programming.

Introduction

Evaluating student responses, particularly natural language responses, is a major challenge in the field of education. It is also a major component of adaptive educational technologies, such as Intelligent Tutoring Systems (ITS), which use automatic evaluation of student responses to maintain an accurate model of student mastery, i.e., a student model. An accurate student model is essential for ITSs for both micro-adaptation, e.g., providing feedback and hints at the step level, as well as for macro-adaptation, e.g., selecting the most suitable instructional tasks for each student to work on next.

However, the assessment of natural language (NL) responses, especially in technical fields like computer programming,is a known challenge. Traditional methods often struggle to capture the semantics of student understanding, particularly when it comes to complex cognitive tasks such as code comprehension. As programming education continues to evolve and scale, there is an increasing need for more sophisticated automated approaches to assessment that can provide accurate and timely feedback to students while alleviating the burden on instructors.

To answer this need, this paper introduces a novel dataset, SelfCode2.0, designed to facilitate the development and evaluation of advanced AI models to automatically assess student NL responses in the context of code comprehension. By creating a comprehensive, annotated corpus of student and expert explanations for Java code, we aim to provide a valuable resource for researchers and educators working on automated assessment tools in programming education. This dataset is timely given the recent advancements in AI and Large Language Models (LLMs), which have heightened the importance of code comprehension skills in modern programming education. Indeed, comprehension is more important now: While Generative AI models can produce code in response to users' NL prompts, the user must comprehend and assess the produced code in terms of correctness and appropriateness relative to their goals. Thus, code writing is diminished for the user/programmer, whereas code comprehension is emphasized.

Code Comprehension and Self-Explanations

Assessing students' code comprehension is central to our research. Code comprehension is vital for both students and professionals who spend more time reading code (others' code or their own code) than writing code. Rugaber (Rugaber 2000) notes that code understanding consumes approximately 70% of the software development lifecycle. Thus, helping students develop code-reading abilities benefits both their academic and professional futures. Self-explanation serves as a key strategy for developing these essential code-reading skills.

Self-explanation is a well-known effective learning strategy that could help students better comprehend texts and master target domains, such as biology. Self-explanations are student-generated explanations of the learning materials (McNamara and Magliano 2009; Crippen and Earl 2004; Van Merriënboer and Sluijsmans 2009; Roy and Chi 2005). Typical self-explanations include inferences based on educational resources and students' knowledge, as well as metacognitive statements, such as students' assessment of their degree of understanding (Roy and Chi 2005). Self-explanations are of personal importance to students and en-

Copyright © 2025 by the authors.
This open access article is published under the Creative Commons Attribution-NonCommercial 4.0 International License.

hance the learning process as they are self-directed and self-generated. Self-explanations involve several cognitive processes, including inferring new knowledge to fill any gaps and combining new knowledge with previously acquired knowledge.

Self-explanation theories (Chi 1989, 2000) indicate that students who engage in self-explanations, i.e., explaining the learning material to themselves while learning, are better learners, i.e., they learn more deeply and show high learning gains. The positive effect of self-explanation on learning has been demonstrated in different science domains such as biology (Chi et al. 1994), physics (Conati and VanLehn 2000), math (Aleven and Koedinger 2002), and programming (Bielaczyc, Pirolli, and Brown 1995; Rus et al. 2021).

Although self-explanation is evidently beneficial, assessing these explanations, especially in programming courses with large enrollments, poses significant challenges. Manual evaluation is time-consuming and often impractical on a large scale, highlighting the need for automated assessment tools. With recent advances in AI and natural language processing, there are exciting opportunities to create these tools; however, their success largely depends on the availability of high-quality annotated datasets for training and assessment.

Currently, there is a lack of annotated datasets that capture both student and expert explanations in the context of code comprehension learning activities. This gap makes it hard for the reliable development of robust and automated assessment models to provide accurate and timely feedback to students on their code understanding. Furthermore, such datasets can be used to evaluate generative AI models, such as LLMs on code explanation tasks and can be included in benchmarks for evaluating the performance of LLMs.

To address this need, we developed the SelfCode2.0 corpus, a novel dataset consisting of self-explanations constructed by students for specific code lines in complete Java code examples, along with corresponding expert explanations. This corpus contains 3,019 sentence pairs of student and expert explanations, each annotated with semantic similarity scores (on a scale of 1-5, 1 being not similar and 5 being most similar), correctness scores (0 or 1 for incorrect / correct), and completeness (0 / 1 for complete or incomplete). The annotations were performed by senior computer science PhD students, ensuring a high level of expertise in the evaluation process. The SelfCode2.0 corpus offers several unique features that make it valuable for advancing automated assessment methods in code comprehension:

- Paired student and expert explanations for a given line of code, useful for direct comparison.
- Annotations based on semantic similarity reflecting four distinct cases of student-expert explanation pairings along with correctness and completeness annotation.
- Focus on line-by-line explanations, allowing for granular analysis of code understanding.

This corpus has significant potential to impact computer science education by enabling more accurate automated assessment tools and the development of ITSs for code comprehension and learning computer programming. These tools could monitor and scaffold students' code comprehension and learning processing by providing timely feedback to students. The paper examines the importance of self-code-type datasets and outlines key steps in collecting and annotating student self-explanations of JAVA code examples. Additionally, it contributes to the advancement of educational technologies for computer science-related skills, which will result in the production of more skilled computer science professionals.

Related Works

Although limited research exists specifically on datasets for code comprehension assessment, our work draws from and contributes to several related areas in computer science education and natural language processing. Various theoretical foundations, including self-explanation theory (Chi 2000; O'Brien 2003), advances in code and text comprehension research (Brooks 1983; Graesser, Singer, and Trabasso 1994; Good 1999; Pennington 1987), and recent progress in automated assessment of open-ended student responses (Banjade et al. 2015) influenced the development of this corpus.

Semantic Similarity in Educational Assessment

Semantic similarity measures have been widely used in various educational contexts to assess student responses. Maharjan et al. (2017) developed a high-performing system for SemEval 2017 by combining an ensemble approach that integrated traditional machine learning algorithms with deep learning models, achieving top-tier performance in the competition. Pontes et al. (2018) developed an innovative neural network model that combines siamese convolutional neural networks (CNNs) and long short-term memory (LSTM) networks. This architecture first employs siamese CNNs to analyze and represent words in their sentential context, followed by siamese LSTMs to process entire sentences based on these word representations and their local contexts. Khayi, Rus, and Tamang (2021) explored the application of fine-tuned pre-trained transformer models for automatically assessing open-ended student answers in physics. They integrated this approach into a conversational Intelligent Tutoring System, demonstrating its potential for real-time, adaptive learning environments. These studies demonstrate the potential for semantic similarity in educational assessment, but mainly focus on more traditional science domains rather than code comprehension specifically.

Code Comprehension in Programming Education

Research on code comprehension in programming education has largely focused on cognitive aspects and teaching methodologies. Schulte et al. (2010) provided a comprehensive review of code comprehension models in novice programming. Letovsky (1986) argued that programmers do not adopt any of the proposed models exclusively; rather, programmers are "opportunistic processors" switching between models as dictated by their knowledge and the task at hand (von Mayrhauser and Vans 1994). Although valuable for understanding the cognitive aspects of code reading, the work of Schulte et al. (2010) and Letovsky (1986) did not extend to creating resources for automated assessment focusing on code comprehension.

Tamang et al. (2021) proposed a structured 10-item rubric for evaluating code comprehension based on code comprehension, self-explanation and other theories. The rubric includes factors such as prior knowledge references, inference making, self-monitoring, control flow understanding, data flow comprehension, and various mental model components. This granular rubric-based approach offers a detailed framework for assessment compared to holistic semantic similarity approaches that compare student explanations against expert references.

Recent work has begun exploring the potential of LLMs for automated assessment of code comprehension. Oli et al. (2024) comprehensively evaluated different LLM-based approaches for assessing students' line-by-line explanations of code, finding that LLMs with few-shot and chain-of-thought prompting can perform comparably to fine-tuned encoder-based models. Additionally, Lekshmi Narayanan et al. (2024) investigated the feasibility of using LLMs to generate and assess code explanations, comparing LLM-generated explanations with those from human experts and students. These studies demonstrate promising directions for using advanced language models to support automated assessment in programming education.

This work contributes distinctly to Oli and Narayanan's (2024) LLM-focused studies by introducing SelfCode2.0, an expert-annotated dataset designed for code comprehension assessment. Rather than evaluating LLM capabilities, we establish baseline performance metrics using TF-IDF and SBERT methods, creating a benchmark dataset that enables future research in automated assessment while avoiding LLM data contamination.

Dataset for Answer Assessment

Although not specific to code comprehension, there have been efforts to create datasets for educational purposes. Rus, Banjade, and Lintean (2014) offered an extensive review of paraphrase or textual similarity identification datasets, encompassing those designed for evaluating student responses. The task of identifying semantic similarity in student responses shares close similarities with computing semantic similarity for assessing open-ended student responses, making this overview particularly relevant to the field of automated educational assessment. SimLex-999 (Hill, Reichart, and Korhonen 2015) is an important dataset that evaluates distributional semantic models that focus on genuine similarity rather than mere association. Their annotation protocol assigned lower scores to word pairs that are related but not truly similar. Banjade et al. (2016) created the DT-Grade corpus, which consists of brief, open-ended responses gathered from student-ITS interactions in the field of Newtonian physics. Their annotation process evaluated the correctness of student responses within the broader conversational context, going beyond isolated comparisons of student self-explanations and expert explanations.

This work addresses a significant gap in the existing literature by building on the work of SelfCode (Chapagain et al. 2023). The original SelfCode corpus was provided as a data set for code comprehension assessment. It has some limitations for multi-sentence explanations as it splits them into individual sentence pairs, potentially losing important context. Our work, SelfCode 2.0, advances this work by preserving the integrity of complete explanations, enabling a more comprehensive assessment of semantic similarity and explanation quality. This improved approach combines elements of semantic similarity assessment, code comprehension, and self-explanation in the context of programming education.

Data Collection and Annotation

Data Collection

This dataset was collected during a lab study conducted in Spring 2022 for introductory Java programming, where students provided line-by-line explanations for code examples through the PCEX interface in the SPLICE catalog (Hicks et al. 2020). Each example included expert explanations from senior computer science PhD students. Students had two attempts to explain each line correctly before receiving a fill-in-the-blank prompt. We collected all first and second attempts, along with the fill-in-the-blank responses from students who failed both initial attempts.

Correctness and Completeness Annotations

Since there are multiple ways of writing correct explanations and the expert explanations are not the only such responses, we need to evaluate if student explanations are ”correct” and ”complete” without relying solely on expert benchmark explanations. To establish a reliable benchmark for evaluating automated assessment systems, we annotated each explanation with correctness and completeness ratings in addition to similarity scores, as detailed below. The goal is to enable the systematic development and evaluation of AI-based approaches for assessing student code explanations.

Correctness and Completeness

Correctness: Student explanation is correct if it explains the behavior or function of the code line correctly. We rated the student explanation as *Correct (1)* or *Incorrect (0)*. Originally, both expert explanations and the students participating in the study were written with the idea of “construction” explanations – why a line of code is needed in the context of the larger program.

Completeness: A student explanation is complete if it explains the code line correctly and covers **all** topics that are necessary and sufficient to explain the line properly. We rate the student's explanation as *Complete (1)* or *Incomplete (0)*. We also used the same labels to mark expert explanations as complete or incomplete, which we used to compare with the corresponding student explanations later.

Similarity Annotations

Including semantic similarity scoring alongside correctness and completeness metrics is crucial for several reasons. First, many automated assessment approaches rely on semantic similarity measures to evaluate student responses against reference solutions. These approaches require reliable human-annotated similarity scores to train and validate the underlying models. Second, semantic similarity captures nuanced variations in student explanations that might be technically correct but expressed differently from expert references. Although correctness and completeness provide some types of assessments, semantic similarity offers a more holistic evaluation capturing both completeness and correctness. Third, in educational contexts where multiple valid explanations exist, semantic similarity helps quantify the degree of alignment between student and expert explanations, enabling more flexible assessment strategies. This is particularly important in programming education, where students may express the same concept using different vocabulary and phrasing while still demonstrating a valid understanding of the underlying programming concepts.

Unlike previous work that fragments multi-sentence responses, our annotation process preserves explanation integrity. We compare each student's complete explanation against individual expert sentences, assessing alignment while maintaining contextual coherence. For each pair of statements produced by experts and students for the same line of code, we rated the semantic similarity between the student and expert explanation on a scale of 1 - 5, with the scale defined as *"highly dissimilar" (1)*, "dissimilar" (2), *"neither" (3)*, *"similar" (4)* and *"highly similar" (5)*. Depending on the number of sentences in the explanation produced by the student and the expert, we accommodated various explanation pairs, leading to four distinct cases:

1. When both the student and the expert provided a single-sentence explanation, the pair receives just one similarity score. For example, [(4)] indicates a high similarity between the single-sentence explanation produced by the student and the expert for the same line of code.
2. If the student provides one sentence while the expert offers multiple sentences, each score represents the comparison of the student's sentence to each expert sentence. For example, [(3,1)] shows varying levels of similarity between expert sentences.
3. When a student provides multiple sentences against a single expert sentence, each student sentence is rated individually. For example, [(4), (1)] indicates how each student's sentence compares to the expert's single sentence.
4. For pairs in which both provide multiple sentences, each student sentence is compared with each expert sentence, resulting in paired scores such as [(2,2), (1,1)].

Table 2 illustrates examples of these cases, demonstrating how different explanation structures were evaluated for similarity. To derive an overall similarity score between a student explanation and an expert explanation from these complex sentence-level ratings, we developed three aggregation methods:

1. Maximum-of-maximum: This approach selects the highest score from all comparisons, highlighting the most significant match between the student and the expert explanations. It provides an optimistic assessment of the student's understanding.

Round	Row Count	Sim.	Corr.	Comp.
1	432	0.871	0.365	-0.012
2	432	0.856	0.263	0.329
3	100	0.797	0.000	0.299
4	100	0.762	0.108	0.275
5	200	0.926	-0.016	0.312
6	200	0.858	0.039	0.648
7	400	0.867	0.103	0.188
8	141	0.814	-0.026	0.117

Table 1: Multiple Rounds of κ Scores for Similarity (Sim.), Correctness (Corr.), and Completeness (Comp.) Annotator Agreement.

2. Maximum-of-average: This method selects the score for the student sentence that aligns most consistently with the expert sentences on average. It also tends towards an optimistic evaluation of students' explanations.
3. Average-of-maximum: This approach computes the mean of the highest similarity scores across sentence comparisons. While it captures the best matches through maximum scores, the averaging process means that weaker alignments in student explanations will lower the overall score. This method provides a balanced assessment by considering both the strengths and weaknesses of a student's explanation.

These aggregation methods were chosen to mitigate what we consider the biggest risk in automated assessment: not giving students credit when they deserve it. Given the current limitations of NLP techniques, there's a possibility that automated methods might incorrectly indicate a student is wrong when they are correct (false negative). Our optimistic approaches aim to minimize this worst-case scenario. Table 3 shows the descriptive statistics of the data set for both the student and the expert explanation

We excluded average-of-averages aggregation as it could underestimate student understanding when a concise, single-sentence explanation matches only one of multiple expert sentences expressing the same concept, unfairly penalizing students for not matching expert sentences.

Annotation Process and Inter-Rater Agreement

One of our goals in developing this dataset is to train a model that automatically evaluates student explanations by comparing them with expert explanations in the current system. To support this implementation, a team of two senior computer science PhD Students annotated the dataset for **Correctness**, **Completeness** and **Similarity**. We conducted multiple rounds of annotation to properly annotate the dataset. In each round, we selected rows to annotate and calculated Cohen's κ scores (see Table 1). For the purposes of annotation, we only considered the student's first attempt explanations in the dataset.

We found that the correctness and completeness annotation was difficult for the raters (see Table 1). We attribute this to the difficulty in differentiating "good" or "bad" stu-

Line of Code	**Student Explanation**	**Expert Explanation**	**Similarity Score**
Point point = new Point();	Creates a class point and creates a variable within that class called point and sets it equal to a method point.	We need to create a Point object to represent a point in the Euclidean plane	[(4)]
private int y;	This declares a private int variable called y	The instance variables are declared as private to prevent direct access to them from outside the class. In this way, no unexpected modifications to a Point object's data are possible.	[(3,1)]
int[] values = {5, 8, 4, 78, 95, 12, 1, 0, 6, 35, 46};	This line creates the integer array with the values. You need this to achieve the goal bc you need an array to look in	We declare an array of values to hold the numbers.	[(4), (1)]
if (num == previous) {	If loop that compares num to previous. If they are equal, then it will print the statement.	We check whether the number that the user entered is a duplicate of the previous number that the user entered. To determine if the two numbers are duplicates, we need to test whether they are equal.	[(2,2),(1,1)]

Table 2: Comparison of Student and Expert Explanations with Similarity Scores

dent explanations. Student explanations are more direct in their description of the line of code when compared with expert explanations – for example, for the line of code *"System.out.println(Maximum value: + maxValue);"*, a student explanation is *"Print out the Maximum value"* expert explanation is *"This statement prints the maximum value in the array to the default standard output stream."*. Hence, it is difficult to distinguish the correct and "good" from the correct but "bad" (lack of understanding) student explanations. This makes it difficult to build a system that evaluates student code comprehension skills and correctness (Oli et al. 2023) as students may also write correct explanations without understanding the code. Therefore, we adopted a more relaxed similarity annotation that led to higher κ scores. We considered a difference of 1 in similarity ratings between two judges as the same rating. For example, if one annotator had rated the similarity as 3 and the other as 2 or 4, we would consider the two ratings the same, inspired by the same approach discussed in SelfCode (Chapagain et al. 2023) when considering a 5-scaled rating of the dataset. There is also some evidence in clinical data annotation over ordinal data rating inter-rater agreements, when "exact" agreement gets the highest "credit" and the scales differing by one get the next highest "credit" and so on (Mitani, Freer, and Nelson 2017). In this way, we could effectively calculate the κ scores (see Table 1).

Explanations	Word Count (μ)	Word Count (σ)
Student	16.41	9.96
Expert	24.78	19.56

Table 3: Statistics of student explanation and expert explanation

We acknowledge that this strategy may sometimes result in more generous scoring than is strictly deserved. However, we believe that this trade-off is appropriate given the current state of technology in educational assessment. The goal is to ensure that students receive due credit for their understanding, even if it means occasionally overestimating their performance. Our detailed annotation process ensures that SelfCode2.0 captures the nuances and complexity of code comprehension explanations, providing a robust foundation for developing and evaluating automated assessment tools in programming education.

Baseline Models: Experiments and Results

Although our primary focus was creating a comprehensive annotated corpus for code comprehension assessment, we also developed baseline methods to serve as benchmarks for future research. These methods provide a foundation for researchers to build upon and compare against when developing more sophisticated automated assessment techniques in programming education, helping to drive innovation in educational technology.

For our baselines, we used Term Frequency-Inverse Document Frequency (TF-IDF) and Sentence-BERT (SBERT) (Reimers 2019). TF-IDF captures lexical overlap between explanations, quantifying term importance in code comprehension contexts. SBERT generates high-quality sentence embeddings that preserve contextual meaning in longer text fragments, used in its pre-trained form without fine-tuning. This combination leverages both traditional and advanced NLP techniques for comprehensive similarity assessment. While specialized architectures like SIAMESE-SBERT might improve performance, our focus was on introducing the SelfCode2.0 corpus rather than model optimization, providing reference benchmarks for future domain-specific research.

Table 4 presents the performance of the two baseline models, TF-IDF and SBERT, across three different aggregation methods: Max_of_Max, Avg_of_Max, and Max_of_Avg. The results are evaluated using Pearson and Spearman correlation coefficients, denoted as P and S, respectively.

Models	Max_of_Max		Avg_of_Max		Max_of_Avg	
	P	S	P	S	P	S
TF-IDF	.310	.290	.313	.293	.313	.293
SBERT	.363	.338	.361	.337	.350	.343

P = Pearson correlation coefficient; S = Spearman correlation coefficient

Table 4: Performance of baseline models on different aggregating methods for semantic similarity.

Both TF-IDF and SBERT implementations used only student and expert explanations as input, excluding code snippets. We calculated cosine similarity between TF-IDF vectors and between SBERT embeddings (using pretrained 'all-MiniLM-L6-v2' without fine-tuning). No hyperparameter optimization was performed. Though our dataset includes correctness and completeness annotations, baseline experiments focused solely on semantic similarity prediction.

The TF-IDF model shows relatively modest performance, with Pearson correlation coefficients ranging from 0.310 to 0.313 and Spearman correlation coefficients between 0.290 and 0.293 across the various aggregation methods. These results indicate that, while TF-IDF captures some degree of similarity between student and expert explanations, its effectiveness is limited compared to more advanced models.

In contrast, SBERT shows modestly improved performance across all aggregation methods, with Pearson correlation coefficients ranging from 0.350 to 0.363 and Spearman coefficients from 0.337 to 0.343. While SBERT better captures semantic relationships in explanations, these results highlight the need for domain-specific fine-tuning. The limited improvement over TF-IDF suggests that, despite SBERT's advanced architecture, its performance could be significantly enhanced with additional training tailored to the code comprehension domain.

Domain-specific fine-tuning would help SBERT better understand programming explanation vocabulary and phrasing. Pre-trained models may not fully grasp the intricacies of specialized language without further adaptation. Bi-encoder models like SBERT need substantial domain-relevant training data to achieve competitive performance, especially with limited or homogeneous datasets.

In summary, while our findings show that SBERT outperforms TF-IDF in capturing semantic similarity, they also underscore the importance of fine-tuning the model on relevant datasets to enhance its applicability and accuracy in automated assessment tasks within programming education. This approach will ensure that SBERT can effectively leverage its capabilities to provide meaningful evaluations of student self-explanations.

While we have implemented baseline models like TF-IDF and SBERT to evaluate the effectiveness of our dataset, our emphasis remains on providing high-quality resources that can be utilized by the community. We believe that by prioritizing the creation of a robust dataset, we enable other researchers to experiment with different models and techniques, ultimately advancing the field of automated assessment in programming education.

Although recently LLMs like ChatGPT have shown promising capabilities in assessment tasks, we deliberately excluded them from our baseline evaluations due to data contamination concerns. Exposing our dataset to these models could compromise future experiments by inadvertently including it in their training data. For this reason, we are making the dataset available only through individual requests rather than public repositories, ensuring its continued value as a reliable benchmark for automated assessment research.

Conclusion

In this paper, we introduced the SelfCode2.0 corpus, a novel dataset designed to enhance the automatic assessment of student self-explanations for code comprehension and beyond. By providing a comprehensive collection of paired student and expert explanations, along with detailed annotations of semantic similarity, we aim to address the existing gap in resources for evaluating natural language explanations of code in programming education. The findings highlight the potential of the SelfCode2.0 corpus to facilitate the development of advanced artificial intelligence models that can provide accurate and timely feedback to students, ultimately improving their understanding of coding concepts.

Future work will focus on developing advanced automated assessment methods by fine-tuning SBERT and other models using domain-specific datasets and using these models to support a ITS to practice code comprehension. We plan to expand our expert explanations by developing comprehensive guidelines for consistent coverage of key programming concepts, while also gathering data from more diverse student populations. These enhancements will improve our dataset's robustness and representativeness, enabling the development of more effective automated assessment tools that can provide accurate, tailored feedback within instructional systems.

Acknowledgments

This work has been supported by the following grants awarded to Drs. Vasile Rus and Peter Brusilovsky (CSEdPad project, NSF awards 1822816 and 1822752, respectively) and to Dr. Vasile Rus (the Learner Data Institute, NSF award 1934745; iCODE, IES awardR305A220385). The opinions, findings, and results are solely those of the authors and do not reflect those of NSF or IES.

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.

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.

Chapagain, J.; Risha, Z.; Banjade, R.; Oli, P.; Tamang, L.; Brusilovsky, P.; and Rus, V. 2023. Selfcode: An annotated corpus and a model for automated assessment of self-explanation during source code comprehension. In *The International FLAIRS Conference Proceedings*, volume 36.

Chi, M. T. H.; Bassok, M.; Lewis, M. W.; Reimann, P.; and Glaser, R. 1989. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science* 13(2):145–182.

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. 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*. 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 *Int. Conference on Intelligent Tutoring Systems*, 304–313. Springer.

Crippen, K. J., and Earl, B. L. 2004. Considering the efficacy of web-based worked examples in introductory chemistry. *Journal of Computers in Mathematics and Science Teaching* 23(2):151–167.

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.; 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.

Khayi, N. A.; Rus, V.; and Tamang, L. 2021. Towards improving open student answer assessment using pretrained transformers. In *The International FLAIRS Conference Proceedings*, volume 34.

Lekshmi Narayanan, A.; Oli, P.; Chapagain, J.; Hassany, M.; Banjade, R.; Brusilovsky, P.; and Rus, V. 2024. Explaining code examples in introductory programming courses: LLM vs humans. In *AAAI 2024 AI for Education Workshop*, volume 257 of *Proceedings of Machine Learning Research*, 107–117.

Letovsky, S. 1986. Emperical studies of programmers, chapter cognitive processes in program comprehension.

Maharjan, N.; Banjade, R.; Gautam, D.; Tamang, L. J.; and Rus, V. 2017. Dt_team at semeval-2017 task 1: Semantic similarity using alignments, sentence-level embeddings and gaussian mixture model output. In *Proceedings of the 11th international workshop on semantic evaluation (semeval-2017)*, 120–124.

McNamara, D. S., and Magliano, J. P. 2009. Self-explanation and metacognition: The dynamics of reading. In *Handbook of metacognition in education*. Routledge. 72–94.

Mitani, A. A.; Freer, P. E.; and Nelson, K. P. 2017. Summary measures of agreement and association between many raters' ordinal classifications. *Annals of epidemiology* 27(10):677–685.

Oli, P.; Banjade, R.; Lekshmi Narayanan, A. B.; Chapagain, J.; Tamang, L. J.; Brusilovsky, P.; and Rus, V. 2023. Improving code comprehension through scaffolded self-explanations. In Wang, N.; Rebolledo-Mendez, G.; Dimitrova, V.; Matsuda, N.; and Santos, O. C., eds., *Artificial Intelligence in Education. Posters and Late Breaking Results, Workshops and Tutorials, Industry and Innovation Tracks, Practitioners, Doctoral Consortium and Blue Sky*, 478–483. Cham: Springer Nature Switzerland.

Oli, P.; Banjade, R.; Chapagain, J.; and Rus, V. 2024. Automated assessment of students' code comprehension using llm. In Ananda, M.; Malick, D. B.; Burstein, J.; Liu, L. T.; Liu, Z.; Sharpnack, J.; Wang, Z.; and Wang, S., eds., *Proceedings of the 2024 AAAI Conference on Artificial Intelligence*, volume 257 of *Proceedings of Machine Learning Research*, 118–128. PMLR.

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.

Pontes, E. L.; Huet, S.; Linhares, A. C.; and Torres-Moreno, J.-M. 2018. Predicting the semantic textual similarity with siamese cnn and lstm. *arXiv preprint arXiv:1810.10641*.

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

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)*.

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*.

Schulte, C.; Clear, T.; Taherkhani, A.; Busjahn, T.; and Paterson, J. H. 2010. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports*. 65–86.

Tamang, L. J.; Alshaikh, Z.; Khayi, N. A.; Oli, P.; and Rus, V. 2021. A comparative study of free self-explanations and socratic tutoring explanations for source code comprehension. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 219–225.

Van Merriënboer, J. J., and Sluijsmans, D. M. 2009. Toward a synthesis of cognitive load theory, four-component instructional design, and self-directed learning. *Educational Psychology Review* 21(1):55–66.

von Mayrhauser, A., and Vans, A. M. 1994. Comprehension processes during large scale maintenance. In *Proceedings of 16th International Conference on Software Engineering*, 39–48. IEEE.