The Thirty-First International Florida
Artificial Intelligence Research Society Conference (FLAIRS-31)

Task-Specific Language Modeling for Selecting Peer-Written Explanations

Eni Mustafaraj, Khonzoda Umarova, Franklyn Turbak, Sohie Lee
Department of Computer Science
Wellesley College, Wellesley, MA
emustafa, kumarova, fturbak, slee@wellesley.edu

Abstract

Students who are learning to program, often write "buggy" code, especially when they are solving problems on paper. Such bugs can be used as a pedagogical device to engage students in reading and debugging tasks. One can take this a step further and require students to explain in writing how the bugs affect the code. Such written explanations can indicate students' current level of computational thinking, and concurrently be used in intelligent systems that leverage "learnersourcing", the process of generating course material for other learners. In this paper, we discuss how to combine learning analytics techniques and artificial intelligence (AI) algorithms to help an intelligent system distinguish between strong and weak textual explanations.

Introduction: Providing Feedback in Computer Science Education

In computer science (CS) education, automatic grading has been present since its beginnings (Hollingsworth 1960). As enrollments in CS courses keep growing, many universities have either created their own sophisticated autograders, or adopted the most successful ones (DeNero et al. 2017). Such systems can assess whether code submitted by students passes the test cases provided by the instructors — that is, whether it is behaviorally correct. However, writing code that behaves correctly is not everything that students should learn. Reading and debugging code are two other important skills that are often neglected in assessments.

Assessing these skills will typically involve asking students to explain in writing (i.e., in plain English) code developed by others, especially code that might contain mistakes (knows as bugs). This kind of assessment, however, poses the familiar scaling problem: what to do with all the written explanations if there are hundreds of students in a class? If assigned frequently, this type of formative assessment could be beneficial to student learning, but expecting personalized, human feedback for such textual answers is infeasible for the size of current CS courses. Therefore, in addition to automated systems to check code quality, we need to build intelligent systems that can check the quality of written explanations about how code works or fails.

Evaluating student writing in other domains is extensively covered by the field of automated essay scoring (Blood 2012). However, essays are a different genre of writing compared to the text of explanatory writing for a CS course. The latter are much shorter, contain domain and task-specific vocabulary, exhibit pervasive misspelling and informal use of language, contain snippets of code, and have a high ratio of ambiguity due to normal English words used as keywords in programming languages (e.g., `for`, `while`, `if`, `else`, etc.). More importantly, in CS writing, correctness matters as much as clarity and conciseness. Meanwhile, providing a score or detailed feedback to improve writing is not a high priority, given the formative nature of the assessment.

The dilemma of how to scale delivery of feedback for student work became more pressing in the context of MOOCs (Balfour 2013). The two big providers, Coursera and edX chose two competing approaches: student calibrated peer review and automated essay scoring, respectively. Both have pros and cons and require known examples of good writing and their scores in advance. For the kind of low-stakes assessment we envision, they have a high entry bar. This is why a more recent approach, known as "learnersourcing" (Kim 2015), might provide a new avenue for providing helpful feedback to students in learning scenarios. Learnersourcing is the process of students generating content for other (future) learners. It has been used in scenarios like generating hints for design problems in engineering (Glassman et al. 2016) or creating subgoal labels for how-to videos (Weir et al. 2015). Closely related to our context, the system AXIS (Williams et al. 2016) collects explanations of math problems written by some confident learners and then displays the best ones to other users of the system. The selection of the best explanations is modeled as a multi-armed bandit problem which makes the system dynamic and adaptable.

Our vision for providing feedback to CS students for their writing involves leveraging correct explanations written by their peers. A feedback-providing system should be able to learn correct, good explanations from the pool of all answers, identify students with incorrect or poorly written answers, and display to them some of the best explanations written by their peers. Our approach toward such a system, described in this paper, combines ideas from learning analytics (Siemens and Long 2011) (e.g. collecting data from learners) and statistical language modeling (Rosenfeld

2000). The language models trained for the debugging tasks assign the best score to concise and correct textual explanations.

This paper is organized as follows: we initially describe the task that leads to generating textual explanations: code debugging, followed by the structure of the system that allows us to collect different kinds of data about the task. The analysis of the collected data sheds light into the kind of decisions we need to make for training the language model, and we show the results of this process. Finally, we discuss the costs and benefits of this approach and provide a roadmap for future work.

The Task: Finding and Explaining Bugs in Code

When students are learning to program a computer, they usually make many mistakes. Initially, these mistakes are syntactic, since it takes some time to learn the syntax rules of a language. Given that code containing syntax errors cannot be executed, students are confronted frequently with their mistakes and learn to fix them. Good integrated development environments (IDEs) provide explanations for syntax errors, pointing in the general area of the error and incorporating some explanation of it, albeit in a language that is often too cryptic for novices. Once students have learned the syntax of a language and have started solving more ambitious problems, they start making semantic errors, errors that reveal their misunderstanding of how a programming language operates. That is to be expected. High-level programming languages, such as Python or Javascript, present a level of abstraction that hides the details of how these languages perform certain actions. It is normal for a learner to require some time to absorb the behavior of different building blocks of a language.

Students' insufficient understanding becomes clear when they are asked to write code on paper, where they don't have the opportunity to try multiple versions of code (trial-and-error) and have to use their initial understanding (or make a guess) about how their code will be executed to solve a given problem. Here is a concrete example for illustration. A programming language like Python allows comparing two strings lexicographically with the relational operators greater than (`>`) and less than (`<`). For example, `'cat'< 'cog'` will be evaluated as `True`, because the comparison is done in the dictionary order. That is, in a dictionary we would find `'cat'` before `'cog'`. Although students have seen examples of how these relational operators act with strings, when asked to solve a problem that integrates this aspect, but is not focused on it, many of them don't think of using the operators in this way. Instead, they decide to write complex code that will compare the characters of the string one by one, trying to also take into account situations where the strings have different lengths. For example, thinking of a comparison such as `'apple'< 'apps'` leads them to code that finds the minimum length of the two strings, so that the index for accessing each character will not try to access an element that doesn't exist. The mistakes that students make in such occasions are a useful source of knowledge for understanding students misconceptions about how to write programs.

In our teaching, we try to put this knowledge to good use. We sometimes create assignment problems that contain so-called bugs (mistakes that usually cannot be attributed to wrong syntax) and engage students in debugging, the process of recognizing and fixing such bugs. To make things realistic, instead of coming up with bugs of our own, we use bugs that students have generated when writing code on paper, or bugs that are inspired by their code. In a typical assignment of this nature, we would explain the given problem, its correct solution, and a series of solutions written by students that contain trivial or non-trivial bugs. The students are then asked to find the bugs and also to explain in writing how each bug affects the code.

While most students might be expected to perform well or to be able to understand the explanations of the instructors (which are made available after students submit their work), a proportion of the students will struggle with the task and not be able to understand the given explanations, pointing to the so-called "expert blind spot": some parts of instructor answers might be inaccessible to students at their current level of understanding (Nathan, Koedinger, and Alibali 2001).

An AI-enhanced system should be able to recognize good answers and weak answers and flag them both as such. Students with weak answers can then be presented with good answers from their peers, as a form of peer instruction (Crouch and Mazur 2001), which is shown to be effective in alleviating the expert blind spot (Wiggins and McTighe 2005).

Learning Analytics: Collecting Data about Learning

The data used in this project was generated in a three-part process: a) a paper exam revealed bugs about a concept; b) a homework assignment asked students to find the bugs and explain them in writing; c) a post-assignment survey asked students to do a self-evaluation of the tasks. The data for parts b) and c) were captured electronically and combined. The dataset contains entries by 159 students, though there is some missing data, because a few students didn't complete all the debugging tasks or didn't fill out the form.

Part A: Revealing Bugs

A good way to reveal students' misconceptions about computation and programming is to ask them to solve problems on paper. Lacking an IDE that allows them to test out countless variations of code composition, when solving a problem on paper they have to use their own understanding of how a program works and how the various building blocks are combined together. In this project, we chose one exam problem in which students had demonstrated that they didn't understand how comparison of string values works. Given a standard filtering problem that had a very simple solution involving a relational operator (as shown in Listing 1), they had created complex solutions that contained bugs of differing complexity. One of the authors cataloged these bugs and created working solutions that typify a bug for pedagogical purposes. One such buggy solution is provided in Listing 2.

Because the bugs revealed other conceptual misunderstandings (e.g., index boundaries, when to break a loop, when to exit a function), it made this problem perfect for the second part of the project: explaining bugs in plain English.

Listing 1: Correct solution for the exam problem given to students to solve on paper.

```
def check(pivot, wordlist):
  result = []
  for word in wordlist:
    if word < pivot:
      result.append(word)
  return result
```

Listing 2: A buggy solution based on solutions written by students during the exam.

```
def check(pivot, wordlist):
  result = []
  for word in wordlist:
    minLen = min(len(word), len(pivot))
    i=0
    while i < minLen and word[i] == pivot[i]:
      i += 1
    if i < minLen and word[i] < pivot[i]:
      result.append(word)
  return result
```

Part B: Collecting Student Explanations

We crafted a homework assignment that contained ten buggy definitions of the `check` function of varying difficulty and asked students to first carefully study the function definitions and then do two things: (1) define minimal counterexamples for each case, that is, a particular set of inputs for which the program will not behave correctly; (2) write brief explanations for why the function is buggy. To prepare them for this problem, which was very unusual, one instructor wrote detailed explanations covering counterexamples, debugging strategies, as well as two solved cases. The entire setup for the assignment amounted to 2500 words including several code samples. The assignment is accessible at this link: https://goo.gl/GtgzRS. Students were given the code in a file, together with variables to store the counterexamples and explanations. They uploaded these files to our server and a script automatically collected their answers. We passed their code through an autograder to check the correctness of their first part: defining minimal counterexamples that identified the bugs.

Part C: Student Self-Evaluation

After submitting their solutions, students were asked to fill out a self-evaluation questionnaire. They were provided with the solutions for all ten buggy problems written by one of the instructors and were asked two questions:

1. Compare your solutions to the provided solutions and then choose the appropriate option, see Figure 1.
2. What debugging strategies did you use for finding the bugs in this problem? Can you list a few of them, providing context for how you used them?

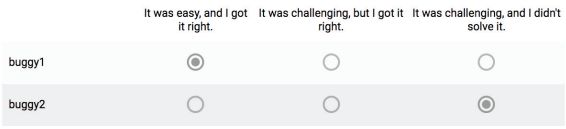

Figure 1: The form for capturing students self-evaluation about how they completed the tasks. The form contains one row for each of the ten problems.

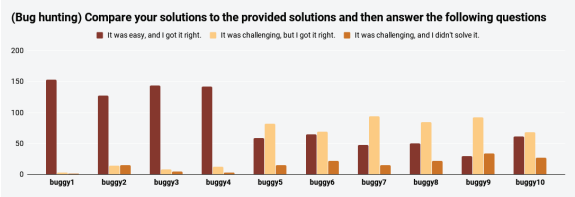

Figure 2: Summary of student self-evaluation for the ten debugging tasks that they solved. The four first tasks were relatively easy, but the difficulty increased and a proportion of students weren't able to solve some of the challenging tasks.

The answers of students for each question are summarized in the bar chart in Figure 2. As previously mentioned, the problems had a different level of difficulty and the self-evaluation of the students clearly indicates that. Some of the problems were challenging for the majority of the class and overall, 72 students didn't solve at least one of the tasks.

The difficulty that students encountered in solving the problems might extend to the difficulty of understanding the solutions. We didn't think of asking students how clear the instructor answers were for them, but one student volunteered that information as part of the self-evaluation form for question two:

> *I noted a lot of my solutions as correct because they had satisfactory counterexamples and I think I understood the bugs, but my explanations were almost nothing compared to those in the solutions, so in that respect, maybe they are not right.*

The uncertainty expressed by this student (who is a strong student) points out to the need for providing some kind of feedback to students, especially to the ones who admitted that they struggled or that they couldn't solve the problem at all. The question is: if the instructor solutions were not satisfactory (because they might suffer from the "expert blind spot"), and reading circa 1500 answers is costly in time, can the explanations provided by other students be used instead? How to find high-quality explanations provided by the students?

Learning Analytics: What does the Data Reveal?

When it comes to assessing writing, the length is often considered a judging criteria. Too short an answer might be insufficient, a very long one might indicate confusion. Thus, we initially looked at the distribution of explanation lengths

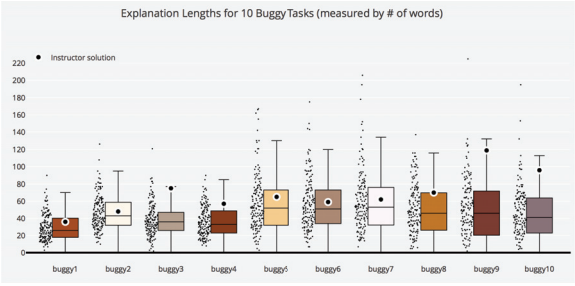

Figure 3: Distribution of explanation lengths as a function of word count for all student submissions. Mirroring the results of task difficulty shown in Figure 2, there is more variability for the explanations of harder questions. The instructor's solutions in the majority of cases were not far away from the median.

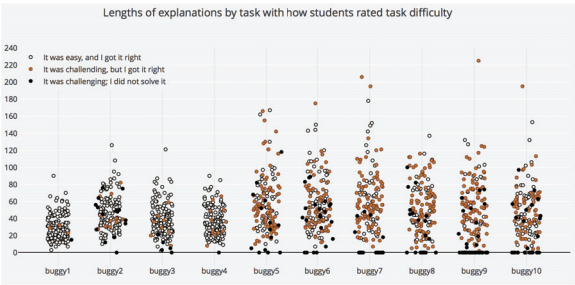

Figure 4: A different representation of the distribution of lengths that colors points by the perceived level of difficulty for the task. No regularity can be seen in the data.

as measured by the count of words. A graphical representation that combines boxplots and scatterplots is shown in Figure 3. We have also included the instructor answer for comparison. As expected, problems that students perceived as simple (the first four ones) show less variability in the explanation lengths, meanwhile, the more challenging questions display higher length variability. That is the first indication that for task-specific writing, the difficulty of the task matters. Any judgment about an individual answer needs to be made in the context of the group of answers for the same task.

Given the variability in the answers length, we were interested in knowing whether the perceived difficulty of the task (according to the self-evaluation) influenced the amount of writing for each student. The graphical representation in Figure 4 colors points by the student's self-reported difficulty level. In terms of length, especially for the challenging tasks, there doesn't appear to be any correlation between perceived difficulty and length of explanations. Even the students who admit of solving the problems wrongly, appear to have written lengthy explanations. Given that the analysis of length variations didn't bring further insights, we decided to look into the word choice and what that might reveal about the answers.

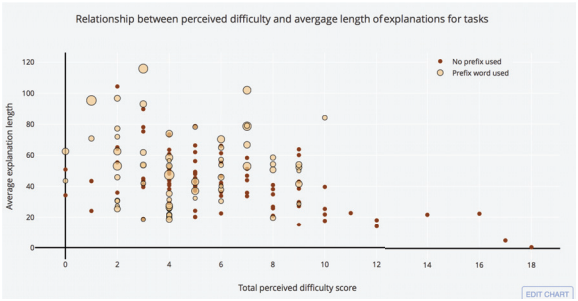

Figure 5: The relationship between the total perceived score of tasks' difficulty and the average answer length. The filled circles show the students who used the word "prefix" in their answers. The size of such circles is proportional to the frequency of using the word "prefix."

Vocabulary Gap Reveals Gaps in Performance

Students in our case study were provided with a 2500-word document (including code) that explained the homework assignment. An entirely new concept featured in this document was that of the "prefix". It was defined in this way: A string p is said to be a prefix of string s if there is some (possibly empty) string q such that $p+q == s$. For example, if pivot is `'dog'`, then the words `'d'`, `'do'` and `'dog'` are prefixes of the pivot, and pivot is a prefix of the words `'dog'`, `'dogs'`, and `'doggy'`. Overall, the word "prefix" was mentioned 14 times in the document, including the comments for the code that students had to debug. However, despite this, when examining the word distribution of students' answers, we discovered that while a few students used the word prefix up to 10 times in their explanations, 78 students didn't use the word at all. Even when they understood that this concept was important in explaining a bug (six of the solutions involved the special treatment of prefixes), students would try to explain it in a round-about way, for example:

> *... the word is shorter than the pivot and has the same letters as pivot until it ends.*

Did the lack of awareness about this important concept, "prefix", affect students' performance? We tested this empirically. We divided students into two groups based on their use of the word "prefix" in their explanations across all ten tasks: 78 didn't use it, 81 used it. Then, we converted the answers for the self-evaluation question shown in Figure 2 into numerical values: 0 - easy, 1 - challenging, 2 - incorrect solution. Each student was assigned a total score between 0-20 based on how they had answered the ten questions. 0 would mean that they had done everything correctly and found the questions easy, 20 that they did everything wrong. We show the relationship between the distribution of these scores and the average length of the explanation across ten tasks in Figure 5. The respective means for the two groups were 4.6 and 6.1 and a t-test for two independent samples rejected the null hypothesis that there is no difference between the two groups with p=0.002. This finding indicates how a single highly-relevant, task-related word can partition a dataset into two groups of students with a different level of

understanding and performance.

In summary of this section, the takeaways from the learning analytics point out to the need to consider the tasks independently of each other, as well as to take into account students' self-reported difficulty.

Statistical Language Modeling

As instructors, our criteria for considering an explanation as high-quality will include several aspects: is the explanation correct? Does it use appropriately the terminology learned in the course? Does it indicate that the student really understood the problem? Then, in addition to these aspects, we may also consider the clarity of the sentences and whether they are well-structured and self-contained.

Human Manual Labeling

In automatic essay scoring, the problem of human subjectivity is well known (Wang and Brown 2007). Thus, we decided to test whether subjectivity plays a role in assessing the short explanations for our debugging tasks. We instructed three raters, one instructor and two teaching assistants, to read 110 explanations and rate them on correctness, clarity/understanding, and to select the best explanation for each task. The answers were randomly chosen from the three groups of students with different levels of self-confidence as well as to contain varying length. The dataset comprised 10 answers by students for each of the 10 tasks and one answer per task written by yet another instructor of the course. The three raters were not aware that the answers contained these previously unseen instructor solutions.

As established in the essay scoring literature, We were also able to confirm that human raters are subjective. We calculated the inter-rater agreement coefficient, Fleiss' Kappa, for the two variables correctness and clarity, and the coefficients were respectively 0.376 and 0.278 (the coefficients are in a scale 0 to 1 with 1 being full agreement). The choice of the best answer also yielded surprising results: the two teaching assistants (TAs) agreed 4 out of 10 times with one another, while the instructor agreed only 2 times with one of the TAs. The instructor and one of the TAs choose 3 times the instructor answer as the best one, the second TA only once.

Why wasn't there much agreement between the raters? We hypothesize that it depends on the apparent similarity of the answers, because they describe the same problems. In fact, several phrases that make up the sentences are almost identical. However, this repetitiveness that "annoys" human graders, is what will make statistical language modeling feasible.

Preparing the Corpus

The text of the explanations written by students differs from normally occurring text in several key ways: a) it contains many characters that are considered punctuation and usually removed from text. However, in the context of programming, all these characters need to be retained, e.g., >=, ==, [ ], etc.; b) it contains snippets of code and/or references to variable and function names, which are English words and introduce ambiguity; c) the so-called functional words (or stop words) are also content words, because they appear as keywords in a programming language. Because of these special characteristics, one cannot simply use out-of-the-box natural language processing tools to prepare the data. This is a cost associated with task-specific text, it has to be handled with tailored tools that take into account its special characteristics. To provide an example, we have included below one explanation written by a student (spelling mistakes left intact), with the only change the font emphasis for task-related words.

> In this case, the bug comes from line 196, the one within the first `IF` statement within the second `FOR` loop. `allLess`, despite given `"TRUE"` before the function is later given `"FALSE"` within the function, therefore screwing everything and making it so nothing is added. If changed to `allLess == True`, then the word `'birb'` would be appended into the return list.

Estimating the Language Model

Statistical n-gram language modeling is a widely used technique in natural language processing that assesses the *fluency of utterances* (Madnani 2009). It is used commonly in applications such as machine translation and automatic speech recognition to identify the degree of goodness for produced sentences: are they likely to occur in everyday language? Concretely, the sentence "She went to the store" is very likely, but the sentence "The store went to her" is unlikely to be uttered. Given the specificity of the task that constraints the students' writing in our scenario, it is possible to get highly likely phrases that indicate consensus, while meaningless phrases will be less common. For example, the trigrams "out of range", "an index error", "is a prefix" occur very frequently in the corpus, because are part of the correct explanations that most students identified, while phrases like "elif will ever" or "the pivot do" occur only once, indicating lack of either correctness or clarity in the explanation.

To estimate the language models (one for each of the ten tasks), we used the package KenLM (Heafield et al. 2013). The combined texts of all student explanations, after the tailored preprocessing to keep as much content as possible, were fed to the KenML package, which outputs the model. We then pass every sentence from each explanation to get its score (the log of the probability value of the sentence). We assigned to every student explanation the average score across all its sentences. In Figure 6, we display the relationship between the length of explanations (x-axis) and the model score averaged by the number of sentences in the explanation (y-axis). As expected, short text will have better scores. When we inspected these short texts, we found that they contained the essence of the solution in a very concise way, for example: "Will not append prefixes to results because allLess will be false if the letters are the same" or "'a' should be appended to the results, but it wouldn't because allLess= False"

What is encouraging from the relationship shown in the graph is that longer explanations have also a good model score, allowing us the to choose answers of different lengths to show to students who might have gotten the solution

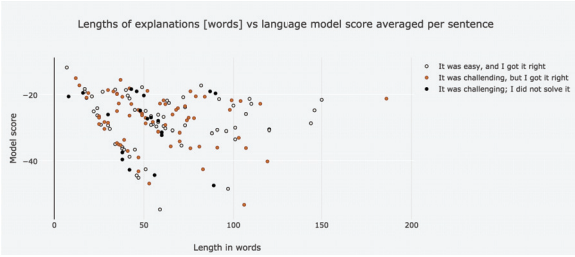

Figure 6: The relation between the length of an explanation and the language model score. This graph represents one task, buggy6. Smaller negative scores indicate more likely sentences than the ones with larger negative scores.

wrong. Finally, let's address the positions of colored dots. It appears that many "incorrect" solutions are shown as very likely. Given that we have access to the counterexamples that the students generated, we calculated the "real" correctness of the task, as opposed to the perceived correctness that students reported. It turns out that only 10 solutions are wrong or missing, with another 27 having chosen a non-minimal instead of a minimal counterexample. Thus, the information provided by the students is somewhat misleading and we are in the process of designing better learning analytics to take into account student's uncertainty about their solutions.

Discussion and Future Work

Assessing students' writing in computer science education is a novel research problem that creates opportunities for combining methods from learning analytics and artificial intelligence. The work presented in this paper shows a promising start for such a combined approach. The language models trained from task-specific written text accurately score the correct solutions as the most likely ones. The fact that the students themselves are more uncertain about their performance than warranted points to the necessity of providing them with results that compare their solutions to those of their peers, together with varying explanations.

We have several ideas for how to continue this work. Initially, we will train models that in addition to the text provided by the students also contain materials from the course, such as homework descriptions or lab explanations. This will bias the model toward more correct formulations. Additionally, we want to introduce more bias to favor certain task-specific concepts, for example, "prefix", in order to rank higher the explanations that use them. Finally, we would like to test the outcome of the system with users and get feedback about the usefulness of providing alternative explanations generated by peers.

References

Balfour, S. P. 2013. Assessing writing in moocs: Automated essay scoring and calibrated peer review (tm). *Research & Practice in Assessment* 8.

Blood, I. 2012. Automated essay scoring: a literature review. *Teachers College, Columbia University Working Papers in TESOL & Applied Linguistics* 11(2):40–64.

Crouch, C. H., and Mazur, E. 2001. Peer instruction: Ten years of experience and results. *American journal of physics* 69(9):970–977.

DeNero, J.; Sridhara, S.; Pérez-Quiñones, M.; Nayak, A.; and Leong, B. 2017. Beyond autograding: Advances in student feedback platforms. In *Proceedings of the 2017 ACM SIGCSE*, 651–652. ACM.

Glassman, E. L.; Lin, A.; Cai, C. J.; and Miller, R. C. 2016. Learnersourcing personalized hints. In *Proceedings of the 19th ACM CSCW*, 1626–1636.

Heafield, K.; Pouzyrevsky, I.; Clark, J. H.; and Koehn, P. 2013. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st ACL*, 690–696.

Hollingsworth, J. 1960. Automatic graders for programming classes. *Communications of the ACM* 3(10):528–529.

Kim, J. 2015. *Learnersourcing: improving learning with collective learner activity*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Madnani, N. 2009. Querying and serving n-gram language models with python. *The Python Papers* 4(2):2009.

Nathan, M. J.; Koedinger, K. R.; and Alibali, M. W. 2001. Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the Third International Conference on Cognitive Science*, 644–648.

Rosenfeld, R. 2000. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE* 88(8):1270–1278.

Siemens, G., and Long, P. 2011. Penetrating the fog: Analytics in learning and education. *EDUCAUSE review* 46(5):30.

Wang, J., and Brown, M. S. 2007. Automated essay scoring versus human scoring: A comparative study. *The Journal of Technology, Learning and Assessment* 6(2).

Weir, S.; Kim, J.; Gajos, K. Z.; and Miller, R. C. 2015. Learnersourcing subgoal labels for how-to videos. In *Proceedings of the 18th ACM CSCW*, 405–416.

Wiggins, G. P., and McTighe, J. 2005. *Understanding by design*. Pearson Education.

Williams, J. J.; Kim, J.; Rafferty, A.; Maldonado, S.; Gajos, K. Z.; Lasecki, W. S.; and Heffernan, N. 2016. Axis: Generating explanations at scale with learnersourcing and machine learning. In *Proceedings of the 3rd L@S*, 379–388. ACM.