A Comparison of the Quality of Data-driven Programming Hint Generation Algorithms

Thomas W. Price, Yihuan Dong, Rui Zhi, *Department of Computer Science*
North Carolina State University
{twprice, ydong2, rzhi}@ncsu.edu

Benjamin Paaßen, *CITEC Center of Excellence*
Bielefeld University
bpaassen@techfak.uni-bielefeld.de

Nicholas Lytle, Veronica Catete, Tiffany Barnes, *Department of Computer Science*
North Carolina State University
{nalytle, vmcatete, tmbarnes}@ncsu.edu

Abstract. In the domain of programming, a growing number of algorithms automatically generate data-driven, next-step hints that suggest how students should edit their code to resolve errors and make progress. While these hints have the potential to improve learning if done well, few evaluations have directly assessed or compared the quality of different hint generation approaches. In this work, we present the QualityScore procedure, a novel method for automatically evaluating and comparing the quality of next-step programming hints using expert ratings. We first demonstrate that the automated QualityScore ratings agree with experts' manual ratings. We then use the QualityScore procedure to compare the quality of six data-driven, next-step hint generation algorithms using two distinct programming datasets in two different programming languages. Our results show that there are large and significant differences between the quality of the six algorithms and that these differences are relatively consistent across datasets and problems. We also identify situations where the six algorithms struggle to produce high-quality hints, and we suggest ways that future work might address these challenges. We make our methods and data publicly available and encourage researchers to use the QualityScore procedure to evaluate additional algorithms and benchmark them against our results.

Keywords. data-driven hints, programming, intelligent tutoring systems, hint quality

INTRODUCTION

Intelligent tutoring systems (ITSs) increasingly use student data to drive their decision making. Rather than relying on extensive knowledge engineering, authors can employ data-driven methods to automate the development of both "outer loop" and "inner loop" (VanLehn 2006) components of an ITS. Examples of this include data-driven construction (Yudelson et al. 2014) and improvement (Koedinger and Stamper 2013) of student models and data-driven hints (Barnes and Stamper 2010) and worked examples (Mostafavi et al. 2015). Authors of data-driven systems argue that these approaches avoid the need for experts to spend time constructing complex domain models (Barnes and Stamper 2010; Rivers and Koedinger 2017) and can lead to additional insights that experts alone would not achieve (Koedinger and Stamper 2013).

In the domain of programming, a growing number of algorithms generate data-driven, next-step hints that suggest how students should edit their code to resolve errors and make progress (Lazar and Bratko 2014; Price et al. 2017d; Rivers and Koedinger 2017). These approaches are promising, as previous work shows the potential of programming hints to improve learning (Corbett and Anderson 2001; Fossati et al. 2015). However, many current evaluations of data-driven hints focus on *whether* a system can reliably provide hints to students (Perelman et al. 2014; Wang et al. 2017) and how much data is necessary to do so (Peddycord III et al. 2014; Rivers and Koedinger 2017). Some prior work has focused on evaluating the *quality* of these hints (Hartmann et al. 2010; Price et al. 2017c,d), but only a few attempts have been made to *compare* algorithms against each other (Piech et al. 2015b; Watson et al. 2012). Such comparisons of hint quality are especially important given prior work showing that the quality of data-driven hints can vary considerably, and that even a single low-quality hint can deter students from seeking help (Price et al. 2017c).

In this work, we present the QualityScore procedure, a novel method to automatically benchmark the quality of next-step, data-driven programming hints by comparing them to expert-authored hints. This allows researchers to evaluate and refine data-driven hint generation algorithms before conducting high-cost user studies. The remainder of this work focuses on two experiments. In Experiment 1, we validate the QualityScore procedure by showing that it agrees with expert ratings of hint quality. In Experiment 2, we use the QualityScore procedure to compare the quality of six data-driven algorithms. Our results show significant, consistent differences in the quality of hints generated by the six algorithms across datasets and problems. We then discuss the challenges uncovered by this analysis and how we might better design algorithms to improve programming hint quality.

DATA-DRIVEN PROGRAMMING HINT GENERATION ALGORITHMS

Data-driven hint generation algorithms leverage data from prior student attempts at a given problem to automatically provide hints to new students attempting the same problem, an approach pioneered by the Hint Factory (Barnes and Stamper 2010). For programming data, we can represent each student's attempt at a problem as a code *trace*, consisting of a series of *code-states* at different points in time, as they progress towards their final, submitted *solution*. These code-states are often stored as abstract syntax trees (ASTs) that represent the syntactic structure of the source code.

When a student requests a hint, hint generation algorithms use their current code-state and trace to provide *personalized* feedback, such as next-step hints, useful example code (Gross et al. 2014a,b), execution trace feedback (Suzuki et al. 2017), recommended test cases (Chow et al. 2017), propagated feedback from instructors (Head et al. 2017; Piech et al. 2015a), or explanations of relevant key concepts (Chow et al. 2017). In this work, we focus on *next-step, edit-based* hints, a common form of support that suggests the next edit a student should make to bring their code closer to a correct solution, such as inserting, deleting or replacing a piece of code.

Hint Generation Algorithms Evaluated in this Work

In this work, we evaluate six next-step hint generation algorithms: TR-ER, CHF, NSNLS, CTD, SourceCheck, and ITAP. Each algorithm takes a training dataset of *correct* solution traces and a set of hint requests as inputs, returning a set of corresponding hints. Each generated hint is represented as a *hint-state*, the new code-state that results from applying the recommended edit to the student's current code-state. Most of these algorithms operate in two phases: first identify a *target code-state* (often a correct solution) in the training dataset, then recommend edits that bring the student's code closer to this target state.

The **Target Recognition – Edit Recommendation (TR-ER)** algorithm (Zimmerman and Rupakheti 2015) defines the target state as the closest correct solution to the student's current code, calculated by the pq-Gram distance (Augsten et al. 2005), an efficient approximation of tree edit distance (Zhang and Shasha 1989). This calculation involves representing an AST as a multiset of all of its subtrees of a specific shape (called pq-Grams). The algorithm then suggests hints to insert, delete or replace code based on the missing or extra pq-Grams in the student's code.

The **Continuous Hint Factory (CHF)** (Paaßen et al. 2018) attempts to define the target state as the next step that an average, capable student would have taken. First, it computes the distance between the student's *trace* (including code history) and the traces of all students in the training dataset who got closer to a correct solution, using a dynamic time warping (DTW) (Vintsyuk 1968) approach[1]. Second, the CHF defines the target state based on predictions of how these capable students would proceed in the student's situation, using Gaussian Process Regression on the DTW distances (Paaßen et al. 2017). Third, the CHF identifies tree edits which bring the student closer to the target state, using a tree grammar to select only valid edits for a given programming language.

The **Next Step of Nearest Learner Solution (NSNLS)** algorithm identifies the closest code-state in the training dataset to the student's current code-state (including incomplete solutions), and defines the target state as the *next* code-state in that trace. It is adapted from work by Gross et al. (2014a), which used tutor-authored code traces. We use student data instead but consider only traces where the next state gets closer to a correct solution. To generate next-step hints from the target state, the NSNLS policy uses the architecture of the CHF, as described in its third step. Therefore, the NSNLS and CHF algorithms differ primarily in how they select the target state.

The **Contextual Tree Decomposition (CTD)** algorithm (Price et al. 2016) was designed to adapt the Hint Factory (Barnes and Stamper 2010) to the domain of programming for use in the iSnap program-

[1]The CHF can also compute distances using only the current code-state of each trace, and we evaluated this version as well. Both versions performed similarly, so we report only the DTW version.

ming environment (Price et al. 2017a). Rather than selecting a single target state from the training dataset, as the Hint Factory does, it decomposes students' ASTs into "subtrees" and matches these smaller pieces of code to those of other students in the dataset. It uses these matches to generate hints at each subtree, taking additional measures to ensure that subtree hints are appropriately contextualized by the code around them. iSnap annotates the student's code with indicators for each hint, allowing the student to select one.

The **SourceCheck** algorithm (Price et al. 2017d) was designed as a replacement for CTD in iSnap (Price et al. 2017a), since CTD sometimes suggested contradictory code from different solutions. SourceCheck selects a single target state, defined as the closest submitted solution to the student's code (as in TR-ER). It calculates this using a code-specific distance metric, which, unlike tree edit distance, can match common code elements from the two ASTs, even when they appear in different locations or orders. SourceCheck suggests hints at each level of the AST based on the differences between the children of these matched code elements. iSnap presents SourceCheck's hints with a combination of hint indicators that the student can select (as with CTD) and code highlighting, as shown in Figure 1.

The **ITAP** ITS (Rivers and Koedinger 2017) generates hints using a five-step process: 1) Canonicalize all code to remove unimportant syntactic variations (Rivers and Koedinger 2013). 2) Identify the closest correct, submitted solution to the student's current code. 3) Apply path construction to identify any closer, undiscovered correct solutions. 4) Identify a target state on the path to the solution that will make a good macro-level hint, as defined by a custom desirability metric. 5) Extract a single edit to present to the student as a hint, usually the edit closest to the root node of the AST. The ITAP system presents this edit as hint text, as shown in Figure 2.

Additional Hint Generation Algorithms

Next-step, data-driven hint generation is still a relatively new research area, and some of the earliest work in the domain of programming was published within the last decade (Jin et al. 2012; Hartmann et al. 2010; Rivers and Koedinger 2014). However, in this time, many approaches have been proposed, in addition to the six we evaluate here. Some of these algorithms attempt to identify a desirable path through a space of previously observed code states, as CHF, ITAP and CTD do. Piech et al. (2015b) developed two "desirable path" hint policies designed to minimize expected completion time and probability of completion, respectively. Others have tried to increase the overlap among students within the state space, for example by using canonicalization (Rivers and Koedinger 2013) or a linkage graph representation (Jin et al. 2012) to reduce syntactic variation, or by representing a student's state using the *output* of their program, rather than its source code (Peddycord III et al. 2014).

Others generate hints using a *cluster* of solutions in the database that match a student's code. Gross et al. (2014b) cluster their database of student solutions into different "solution strategies," identify a cluster for the hint-requesting student, and generate hints using a "prototype" solution for that cluster. The MistakeBrowser (Head et al. 2017) identifies clusters of code states that have the same problem (e.g., failing a unit test) and uses program synthesis to propagate one student's fix to others in the cluster. Instructors can annotate these fixes with additional feedback. Similarly, Chow et al. (2017) match the hint-requesting student to a cluster of code-states that failed similar test cases and have similar structure, and recommend edits that the *majority* of these students took.

Like the TR-ER and SourceCheck algorithms, the iGrader system (Wang et al. 2017) attempts to identify the closest *submitted, correct* solution to use for hint generation, using a distance metric similar to tree edit distance. Like the CHF, the DeepFix (Gupta et al. 2017) and sk_p (Pu et al. 2016) systems use machine learning (specifically neural networks) on encoded code-states to generate hints. Similarly, Piech et al. (2015a) encode code-states using linear functions which map inputs to outputs, and they propagate instructor feedback based on a classification of these linear functions. Like the CTD algorithm, some approaches attempt to break a program down into smaller pieces and generate hints for the pieces individually. Lazar et al. proposed breaking programs into individual lines of code to generate hints for each line (Lazar and Bratko 2014). They later proposed a rule-based approach to hint generation, where they extracted common AST patterns from a dataset of both correct and incorrect student attempts, and generated hints to add missing correct patterns or remove problematic patterns (Lazar et al. 2017).

The six algorithms we selected for comparison in this work were chosen because, unlike the other algorithms in this section, they: 1) generated data-driven next-step hints, 2) had clear implementation details or available source code, and 3) required no unit tests or assumptions about the connectedness of the code-state space to operate (since our datasets did not have these properties). Our goal is not to produce a comprehensive comparison of all next-step hint generation algorithms, but rather to demonstrate how such a comparison can be used to gain insight. While we excluded some of the above algorithms due to the second criterion (Chow et al. 2017; Head et al. 2017; Lazar and Bratko 2014; Piech et al. 2015b) and the third criterion (Jin et al. 2012; Pu et al. 2016; Wang et al. 2017), with some modification and available code, they could still be evaluated by the QualityScore procedure. Our hope is that others will do so, and compare their results to the six algorithms evaluated in this work. Other data-driven systems are not comparable to those evaluated here, failing our first criterion because they give feedback other than next-step, edit-based hints (Gross et al. 2014b; Lazar et al. 2017; Peddycord III et al. 2014; Piech et al. 2015a). Others focus on resolving specific syntax or runtime errors (Gupta et al. 2017; Hartmann et al. 2010; Watson et al. 2012), rather than semantic errors, or on improving the style of already correct programs (Choudhury et al. 2016; Moghadam et al. 2015). Evaluating these algorithms is beyond the scope of this work, but we hope that similar, rigorous measures of hint quality can be developed for these hints as well.

Evaluations of Next-step Programming Hints

We have previously identified six types of evaluations of automated and data-driven programming hints (Price et al. 2018): availability, cold start, correction, student choice, student outcomes, and expert evaluations. The first four are primarily technical evaluations, showing whether hints *can* be generated (Perelman et al. 2014), how much data is necessary to do so (Chow et al. 2017; Rivers and Koedinger 2017), how often they fully correct erroneous code (Gupta et al. 2017; Lazar and Bratko 2014; Wang et al. 2017), and how well they predict students' later progress on a problem (Lazar et al. 2017; Price et al. 2016), respectively. These evaluations only indirectly address hint quality, since even hints that lead to a correct solution may not be easily understood (Price et al. 2017a,b).

Student outcome evaluations are ideal for assessing learning, but they are also rare and costly to implement. The iList tutor (Fossati et al. 2015), which provides feedback including data-driven, next-step hints, led to learning gains comparable to working with a human tutor. Choudhury et al. (2016) found that their data-driven style feedback for already-correct programs, which included next-step hints, helped students achieve significantly higher-quality solutions than a control group. Students in the ACT Programming tutor (Corbett and Anderson 2001) who received any form of feedback, including next-step hints, completed the tutor faster and completed a later assessment faster and with fewer errors than a control group. In contrast, Yi et al. (2017) found that their automated hints *increased* the time it took novice students to complete a set of debugging tasks, compared to a control group with no hints. Importantly, most of these studies evaluated next-step hints *along with* other feedback, and none of them evaluated students completing entire open programming tasks.

Expert Evaluations directly measure the quality of data-driven hints using experts, which are useful for evaluating and improving algorithms before investing in a student evaluation. Some expert evaluations ask experts to rate hints directly on a validity scale (Hartmann et al. 2010; Price et al. 2017c; Watson et al. 2012). Others compare expert judgements to those of the algorithm. Gross et al. (2014b) used this approach to evaluate their algorithm's ability to effectively cluster student code for feedback generation. Piech et al. (2015b) asked a group of experts to generate single, "gold standard" hints for a set of student code-states, and they evaluated hint systems based on their accuracy in matching these gold standard hints. In our prior work (Price et al. 2017d), we extended this gold-standard approach by having experts identify a *set* of valid hints, rather than a single best hint, and by generating hints for actual student hint requests.

METHODS

In this work, we present a novel QUALITYSCORE procedure to benchmark the expert-perceived quality of data-driven programming hints, and we use it to investigate four research questions:

RQ1: How well do QUALITYSCORE ratings agree with manual expert hint ratings?

RQ2: Which data-driven programming hint generation algorithms have the highest-quality hints on our datasets?

RQ3: How do state-of-the-art programming hint generation algorithms compare to human tutors?

RQ4: Under what circumstances do current state-of-the-art programming hint generation algorithms fail to produce high-quality hints?

Datasets

To investigate our RQs, we analyzed datasets from two programming environments that offer on-demand, data-driven, next-step hints: iSnap (Price et al. 2017a), a block-based programming environment, and ITAP (Rivers and Koedinger 2017), an intelligent tutoring system for Python programming. Figure 1 and Figure 2 show how next-step hints are presented in both systems. Both datasets consist of log data collected from students working on multiple programming problems, including complete traces of their code and records of when they requested hints. The iSnap dataset was collected from an introductory

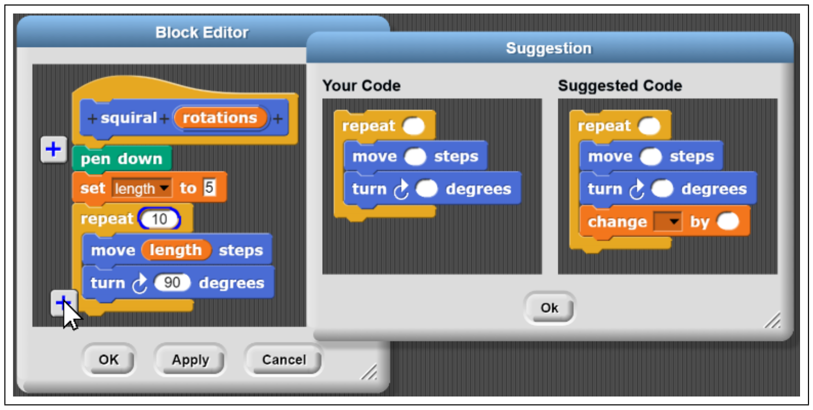
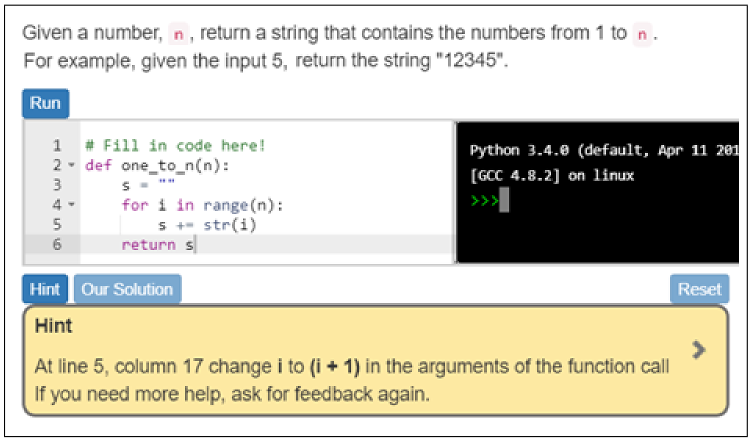


Fig.1. A next-step hint presented by iSnap on the Squiral problem.



Given a number, n, return a string that contains the numbers from 1 to n.
For example, given the input 5, return the string "12345".

```
# Fill in code here!
def one_to_n(n):
    s = ""
    for i in range(n):
        s += str(i)
    return s
```

```
Python 3.4.0 (default, Apr 11 201
[GCC 4.8.2] on linux
>>>
```

Hint

At line 5, column 17 change **i** to **(i + 1)** in the arguments of the function call
If you need more help, ask for feedback again.

Fig.2. A next-step hint presented by ITAP on the OneToN problem.

programming course for non-majors during the Fall 2016, Spring 2017, and Fall 2017 semesters, with 171 total students completing six programming problems. Fall 2016 and Spring 2017 data have been analyzed in our prior work (Price et al. 2017c,d, 2018). The Python dataset was collected from two introductory programming courses in Spring 2016, with 89 total students completing up to 40 Python problems (see Rivers et al. 2016, for details). Both datasets are available[2] from the PSLC Datashop (pslcdatashop.org) (Koedinger et al. 2010).

Our evaluation of data-driven hint quality required two sets of data: a set of *hint requests*, and a set of *training data* used to generate the hints. From the iSnap dataset, we randomly sampled one hint request per problem from each student who used hints. This ensured that no student was overrepresented in the set of hint requests. From the Python dataset, we randomly sampled up to two unique hint requests from each student, since there were fewer students who requested hints than in the iSnap dataset. We only

[2]All datasets and procedures used in this study are available at: `go.ncsu.edu/hint-quality-data`.

sampled hint requests where the student's Python code compiled, since this is required by most of the algorithms we compare. We also extracted a set of training data from each dataset, consisting of correct solution traces from students with no hint requests. Although students in both the iSnap and Python datasets received hints (from SourceCheck/CTD and ITAP, respectively), we analyze all hint requests *before* the hints were delivered to minimize the impact of these hints on our analysis.

From the iSnap dataset, we selected two representative problems to analyze, GuessingGame and Squiral, which have been used in previous evaluations of iSnap (Price et al. 2017d,c). The two problems had 31 and 30 hint requests, respectively (61 total). Common solutions to these problems are approximately 13 and 10 lines of code, respectively, and require loops, conditionals, variables and procedure definitions. From the Python dataset, we selected the 5 problems that had the most hint requests, for a total of 51 hint requests (7-14 per problem). These simpler problems had common solutions with 2-5 lines of code, which included variables, API calls, arithmetic operators and, for one problem, loops. An important difference between the datasets is that the iSnap problems are considerably longer and more open-ended, while the Python problems often involve single-line functions that can be evaluated with test cases.

The QualityScore Procedure

We evaluate the quality of next-step, data-driven[3] hints by comparing them to a set of "gold standard" hints, authored by a group of three *tutors* familiar with the problems. We used a two-phase process to generate the gold standard hints, and we carried this process out independently for the iSnap and Python datasets, using two different groups of tutors (in this case, the paper authors served as tutors).

In **Phase I**, the three tutors independently reviewed each hint request, including the history of the student's code before the hint request, and generated a complete set of next-step hints that met the following *validity criteria*: All hints should be *relevant* and *useful* given the student's current code and history. They should be as *minimal* as possible while remaining *interpretable*. Each hint should be *independent*, requiring no additional hints or explanation. Each hint was represented as one or more edits to the student's code, and tutors could also include *blanks* (_) in their hint to leave part of it unspecified (e.g., "add `x + _`"), for the student to figure out.

In **Phase II**, each tutor independently reviewed the hints generated by the other two tutors and marked each hint as valid or invalid by the same validity criteria used above. Our gold standard set includes all hints considered valid by at least two out of three tutors. Our goal was not to determine a definitive, objective set of correct hints but rather to identify a large number of hints that a reasonable human tutor might generate. Having two tutors agree on each hint provides a higher quality standard than what is used in most classrooms, while also capturing a diversity of hints. This produced between 1 and 11 gold standard hints per hint request for the iSnap dataset (Med = 5) and between 1 and 5 for the Python dataset (Med = 2). For the iSnap dataset, tutors met again to resolve disagreements, producing a set of *consensus hints* that we used in Experiment 1.

We use the set of gold standard hints produced in Phase II to automatically assign a QualityScore to a hint generation algorithm A for a set of hint requests R, as shown below in Equation 1. For each

[3]While the QualityScore was especially designed for data-driven hints, it can be used to evaluate any next-step hint.

hint request $r \in R$ we use the algorithm to generate a set of candidate hints, $H_{A,r}$. The algorithm must also assign a confidence weight w_h to each hint h it generates, with weights summing to 1. We calculate the validity of each hint $V(h)$ as 1 if it matches a gold standard hint ($h \in G_r$) and 0 otherwise. The QUALITYSCORE for a given hint request r is the sum of the weights of all valid hints in $H_{A,r}$, ranging from 0 (no valid hints) to 1 (all valid hints). We average this value over all hint requests in R.

$$\textsc{QualityScore}(A, R) = \frac{1}{|R|} \sum_{r \in R} \sum_{h \in H_{A,r}} w_h \cdot V(h); \quad \text{where } V(h) = \begin{cases} 1 & \text{if } h \in G_r \\ 0 & \text{if } h \notin G_r \end{cases} \tag{1}$$

To determine if a candidate hint h matches a gold standard hint g, we first normalize their respective ASTs by standardizing the names of any newly-created variables, methods and string literals and by removing compiler-generated AST structures (like default child nodes). We used two thresholds for determining if h matches g: A *full match* occurs when h and g are identical, while a *partial match* occurs when h suggests exactly a subset of the edits in g, and at least one of these edits is an insertion. We can calculate the QUALITYSCORE counting only full matches as valid, or counting partial matches as valid also (which yields a higher score). Both QUALITYSCORE values give insight into the quality of an algorithm.

The QUALITYSCORE procedure is capable of evaluating algorithms that generate multiple hints, as many do, which allows the algorithm multiple opportunities to generate valid hints. However, algorithms are also penalized for generating invalid hints, even if they are accompanied by valid ones. We believe that this is reasonable, since low-quality hints have been shown to deter students from future help-seeking (Price et al. 2017c). This raises the question of which hint is actually *shown to the student*. In this work, we distinguish between the hint generation algorithm, responsible for generating and prioritizing hints, and the help interface, responsible for choosing which hints to show and how. Some help interfaces, such as iSnap's, annotate a student's code with indicators for all available hints and allow the student to select one (see Figure 1), while others, such as ITAP's, show only a single, highest-priority hint (see Figure 2). We could imagine other approaches, such as choosing which hints to show adaptively, based on a student model. The QUALITYSCORE evaluates an algorithm based on its ability to generate valid hints and weight them to inform a help interface's choice of what hints to show, regardless of the interface.

EXPERIMENT 1: VALIDATING THE QUALITYSCORE PROCEDURE

Previous evaluations of programming hints have used human experts to *manually* rate hints as valid or invalid (Paaßen et al. 2018; Watson et al. 2012). A primary advantage of the QUALITYSCORE procedure over this manual expert rating is that it is *consistent* and *replicable*, meaning that we can automatically and objectively apply it to any number of hint generation algorithms, without requiring additional expert time. In this experiment, we investigate RQ1, which asks whether the QUALITYSCORE ratings are consistent with manual hint ratings.

Procedure

In this experiment, we selected a set of automatically generated hints from the iSnap dataset and rated these hints both with the QUALITYSCORE procedure and manually with experts. High agreement between these two ratings would suggest that QUALITYSCORE is a reasonable substitute for manual hint ratings and accurately reflects expert judgement. Our goal was to evaluate this agreement on a diverse set of data-driven hints, with different levels of hint quality. We used four hint generation algorithms (Zimmerman, CHF, CTD, and SourceCheck)[4] to generate these hints for each hint request in the iSnap dataset. We used the QUALITYSCORE procedure to rate each hint as valid (matches a valid hint), partially valid (matches a subset of the edits in a valid hint) or invalid (matches no valid hints). The QUALITYSCORE rating marked over 80% of all generated hints as invalid, so in order to avoid over-representing invalid hints in our evaluation, we created a more balanced sample as follows: For each hint request, we randomly selected 1 hint per algorithm (if present) that had a QUALITYSCORE rating of valid, partially valid, and invalid, respectively, resulting in 119 valid, 84 partially valid and 233 invalid hints. We then randomly sampled 84 of each type for a balanced sample of 252 total data-driven hints.

The same three human tutors who created the gold standard hints for the iSnap dataset then manually rated the data-driven hints. First, each tutor individually viewed each of the 252 hints, along with the respective hint request and code history, and assigned each hint a rating of valid, partially valid, or invalid. The tutors were told to use the same validity criteria that they used to rate each other's hints in Phase II of the gold standard hint generation. They were instructed to rate a hint as "partially valid" if all of its suggested edits were useful and important, but a student would need additional information to interpret the hint. For example, imagine a student who has used a for-loop where they should have used a while-loop instead. A data-driven hint that suggested this replacement would be "valid," but one that simply suggested adding the while-loop (without replacing the for-loop) might be rated "partially valid," since that is useful but incomplete information. The pairwise agreement between the tutors, as measured by weighted Cohen's kappa, ranged from 0.65 to 0.69, indicating substantial agreement, and the three tutors reached perfect agreement on 166 out of 252 hint requests (65.9%). Any disagreements were discussed by the tutors until a consensus validity rating was reached for all 252 hint requests.

Results and Discussion – RQ1

To address RQ1, we calculated the agreement between the automated QUALITYSCORE hint ratings and the manual tutor ratings. The weighted Cohen's kappa was 0.78, indicating substantial agreement[5], with the QUALITYSCORE matching 82.9% of the manual hint ratings exactly. As described earlier, the QUALITYSCORE procedure includes any hint approved by at least 2 out of 3 tutors in Phase II in the gold standard hint set. Recall that on the iSnap dataset, we also had tutors discuss and come to consensus on each gold standard hint. We investigated whether this consensus phase offers any advantage by calculating the QUALITYSCORE using this consensus gold standard instead. Doing so raises the kappa

[4]Our implementation of the NSNLS algorithm was completed after this experiment, and the ITAP algorithm only operates on the Python dataset, so we did not include them.

[5]Originally the kappa was 0.64, which prompted us to change the way the QUALITYSCORE identifies partial matches (requiring at least one overlapping insertion), and this substantially improved agreement.

value slightly to 0.81. Similarly, counting *all* hints generated by any tutor in Phase I as gold standard hints (without any tutor Phase II agreement) produces a kappa of 0.83. We conclude that the choice of gold standard criteria simply biases the QUALITYSCORE to be more or less permissive, but it does not meaningfully change the overall agreement with manual hint ratings. For comparison, the agreement between the three tutors' individual manual ratings and their eventual consensus ratings was 0.85, 0.76 and 0.78, respectively. This indicates that the QUALITYSCORE matches the consensus ratings about as well as any individual tutor.

We conclude that the QUALITYSCORE generally captures our human tutors' understanding of hint quality. However, the agreement was not perfect, so we also manually inspected the 43 hints where the QUALITYSCORE disagreed with the manual hint ratings. We found that the majority of these disagreements (67%) involved a rating of "partially valid," indicating that the QUALITYSCORE procedure may have some trouble distinguishing partially valid hints from valid and invalid hints. For the remaining hints, almost all disagreements involved substantial discussion among the human tutors. Often the tutors even changed their minds, for example including a hint in the gold standard during Phase II, but rating the *same hint* as invalid during Experiment 1, or vice versa. This underscores the subjective nature of human hint quality ratings. Our QUALITYSCORE reflects those same subjective tutor judgements, but unlike human tutors, it is *consistent* in applying those judgements to different algorithms. While there are other, valid ways to assess hint quality (e.g., asking students to rate hints, or evaluating hints' impact on learning outcomes), with the QUALITYSCORE procedure, we can easily evaluate and compare hints from any number of algorithms, as we do in the following section.

EXPERIMENT 2: COMPARING ALGORITHMS

In this experiment, we investigated RQ2 and RQ3, evaluating how the quality of data-driven hint generation algorithms compared with each other and with human tutors.

Procedure

We used the QUALITYSCORE procedure to compare the six data-driven next-step hint generation algorithms described in detail earlier: TR-ER, CHF, NSNLS, CTD, SourceCheck and ITAP. We evaluated each algorithm on the 2 assignments in the iSnap dataset and 5 assignments in the Python dataset. For each assignment, we trained the hint generation algorithm on the training dataset and then calculated the QUALITYSCORE on the full set of hint requests.

As described earlier, the QUALITYSCORE procedure requires that each algorithm assign a *confidence weight* to each hint that it generates, where higher-weighted hints contribute more to the final QUALITYSCORE. The CHF and NSNLS algorithms associate an error with each hint they generate, representing how far the hint-state is from the target state, which is translated into a weight[6], such that higher errors correspond to lower weights. SourceCheck uses a voting-based approach that assigns weights to hints based on their representation in the top-k closest matching target solutions (Price et al. 2018).The

[6]The weight w_i for a hint with error e_i is given by $w_i = \exp(-(e_i - \min(E))/(\bar{E} - \min(E)))$, where E is the set of all generated hint errors for a given hint request and $\bar{E}$ is the average error.

ITAP algorithm generates only one hint per hint request, which is given a confidence weight of 1. The TR-ER and CTD algorithms present no method for weighting the importance of hints. Rather than attempting to define a reasonable weighting approach ourselves, we assigned each hint a uniform weight. While we acknowledge that this uniform weighting is likely not optimal (c.f. Price et al. 2018), we argue that it fairly reflects that algorithm's current inability to prioritize hints, which represents an important area for future work.

For most of the algorithms we evaluated, we used an implementation written by the original paper authors, and we adapted these implementations to use our training data and to output code-states compatible with our gold standard hints. One exception is the NSNLS algorithm, where we base the implementation off of the CHF and use student data rather than the expert-authored traces used by the original authors (Gross et al. 2014a). The other exception is the ITAP algorithm which is designed specifically for the Python programming language and cannot operate on the generic ASTs in our training dataset. ITAP also requires a set of unit tests to operate most effectively, which our dataset did not include. Instead of generating a new set of ITAP hints, we used the historical hints that the ITAP algorithm originally generated when the Python dataset was collected. We were therefore only able to evaluate the ITAP algorithm on the Python dataset, and these hints are different in a few key ways. First, since ITAP built up its training dataset over time as solutions were submitted throughout the study, each hint was generated using a different subset of the total training data. This training data may have also included students who requested hints, which our training dataset does not. Second, ITAP used a Python-specific canonicalization process (Rivers and Koedinger 2013), which none of the other algorithms could do, since they had to operate on generic ASTs. Third, ITAP used unit tests for path construction, which were unavailable to the other hint generation algorithms. Lastly, ITAP's hints were conveyed in natural language, rather than as AST edits. In order to represent these as code-states, we manually carried out each hint that ITAP described and saved the resulting Python AST as the hint-state. This process guarantees that the resulting hints are syntactically valid, making them easier to match to the gold standard hints. These differences mean that the comparison between ITAP and the other hint generation algorithms is not exact, and may favor ITAP, since it is able to use Python-specific features, unit tests, and hand-authored hint-states.

Results

Figure 3 shows the QUALITYSCORE ratings for each algorithm on the iSnap and Python datasets. As a reference, the figure also shows the QUALITYSCORE of the Tutors, the original hints produced by all the human tutors in Phase I, before the hints were rated and filtered to make the gold standard set. This gives an upper baseline for the QUALITYSCORE for a given assignment. Recall that the QUALITYSCORE can be calculated to count only hints that have a *full match* to a gold standard hint as valid, or to also count *partial matches*, which have a meaningful subset of the edits. The figures report the QUALITYSCORE calculated with both full and partial matches, but since they are similar, in the remaining analysis we focus on full matches unless otherwise stated.

There is a clear difference between the QUALITYSCORES of the six algorithms, and a Kruskal-Wallis test confirms that this overall difference was significant for the iSnap dataset ($H(4) = 63.5$; $p < 0.001$). We performed post hoc, pairwise Wilcoxon signed-rank tests, using the Benjamini-Hochberg

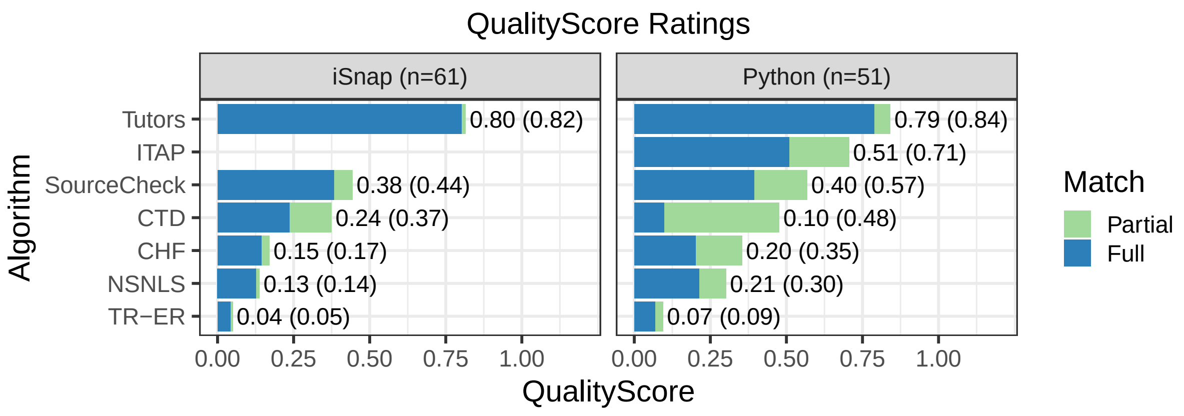


Fig.3. QUALITYSCORE ratings for each algorithm and the human tutors on the iSnap and Python datasets, calculated with full matches only (blue) and including partial matches (green). Labels indicate full (and partial) QUALITYSCORES. Ratings are averaged over all n hint requests in the dataset.

procedure (Benjamini and Hochberg 1995) to control the false discovery rate at 5%. We find that TR-ER < NSNLS = CHF < CTD < SourceCheck < Tutors on the iSnap dataset. These trends hold when including partial matches, with the exception that there is no significant difference between CTD and SourceCheck. The highest-performing algorithm, SourceCheck, performed 47.9% and 54.4% as well as the human tutors, without and with partial matches, respectively.

For the Python dataset, we also see a significant difference between the QUALITYSCORES of the algorithms (Kruskal-Wallis test; $H(5) = 35.4$; $p < 0.001$). The overall ranking of the algorithms is similar to the iSnap dataset, with the addition of the ITAP algorithm, which performs best. Post hoc pairwise Wilcoxon signed-rank tests show that TR-ER = CTD < CHF = NSNLS < SourceCheck = ITAP < Tutors on the Python dataset. However, we do see differences in the ranking of the algorithms on the Python dataset when including partial matches: TR-ER < NSNLS = CHF < CTD = SourceCheck < ITAP = Tutors. This second ordering is consistent with the iSnap dataset. The highest-performing algorithm, ITAP, performed 64.5% and 83.9% as well the human tutors, without and with partial matches.

Figures 4 and 5 show the QUALITYSCORE ratings for each assignment in the iSnap and Python datasets, respectively. The two iSnap assignments follow similar trends, with the algorithms achieving similar scores and rankings on both assignments. The Python dataset features 5 simpler assignments, and there are larger differences in the QUALITYSCORES and ranks of the algorithms across assignments. However, the ITAP algorithm consistently performs best across assignments, approaching the human tutors' QUALITYSCORE on some. Many algorithms also perform much better when including partial matches in the QUALITYSCORE on the Python dataset, such as on FirstAndLast and KthDigit, where four algorithms more than double their QUALITYSCORE with partial matches.

The QUALITYSCORE procedure can also be used to calculate detailed statistics about the performance of each algorithm, which are shown in Table 1. We see that the algorithms with the highest QUALITYSCORES are also generally more likely to produce *any* valid hint for a given request, and they

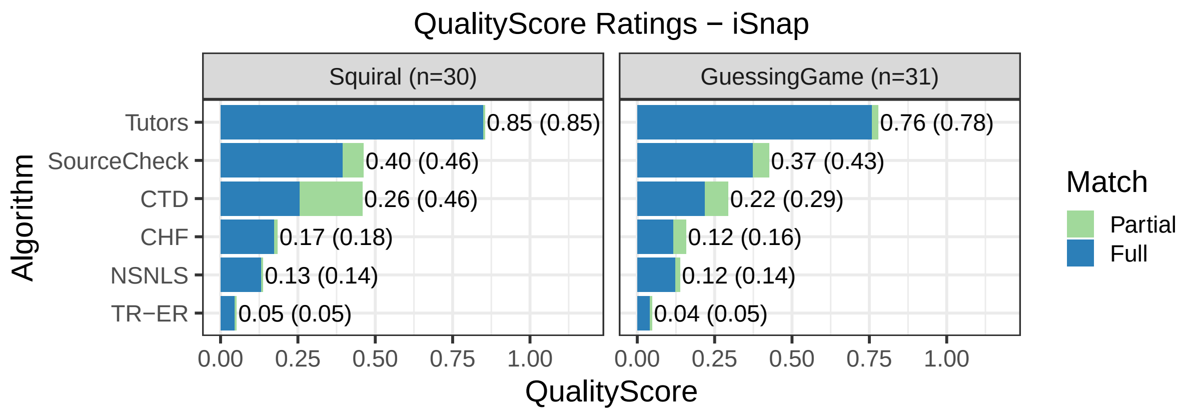


Fig.4. QUALITYSCORE ratings for each algorithm on each assignment in the iSnap dataset.

generally produce more valid hints per hint request on average, suggesting that the QUALITYSCORE reflects these metrics of hint quality as well. However, there is a less clear relationship between the QUALITYSCORE and the number of hints an algorithm generates per hint request. For example, while both ITAP and SourceCheck had a high QUALITYSCORE on the Python dataset, SourceCheck generated on average over twice as many hints, resulting in both more valid hints on average, and also more invalid hints. It is important to consider these different aspects of the quality of a hint generation algorithm, especially when trying to diagnose problems with an algorithm (as we discuss in the following section). However, it can also be difficult to reconcile multiple quality metrics when comparing algorithms, or when determining whether changes to an algorithm produced an improvement. The QUALITYSCORE addresses this by capturing many meaningful properties of hint quality in a comprehensive value.

Discussion – RQ2 and RQ3

The SourceCheck algorithm performs significantly better than all others on the iSnap dataset, and the ITAP and SourceCheck algorithms perform significantly better than all others on the Python dataset (not counting partial matches). Importantly, SourceCheck was originally designed to generate hints for iSnap, and ITAP was designed specifically for Python, which suggests that one element of these algorithms' success is their attention to context-specific and programming-language-specific aspects of hint generation. This may partially explain why the TR-ER and NSNLS algorithms performed poorly, since the other algorithms had been previously evaluated using programming logs from either iSnap (CTD, SourceCheck, CHF: Price et al. 2016, 2017d; Paaßen et al. 2018) or Python (ITAP: Rivers and Koedinger 2017). However, more than dataset-specific factors contributed to hint quality, since SourceCheck performed quite well *across* datasets, and all algorithms ranked similarly on both datasets.

There were clear differences in the performances of the algorithms, with the top-ranked algorithm performing 9 and 7 times better than the bottom-ranked algorithm on the iSnap and Python datasets, respectively. Figure 3 shows a wide range of hint quality across the algorithms on both datasets, suggesting

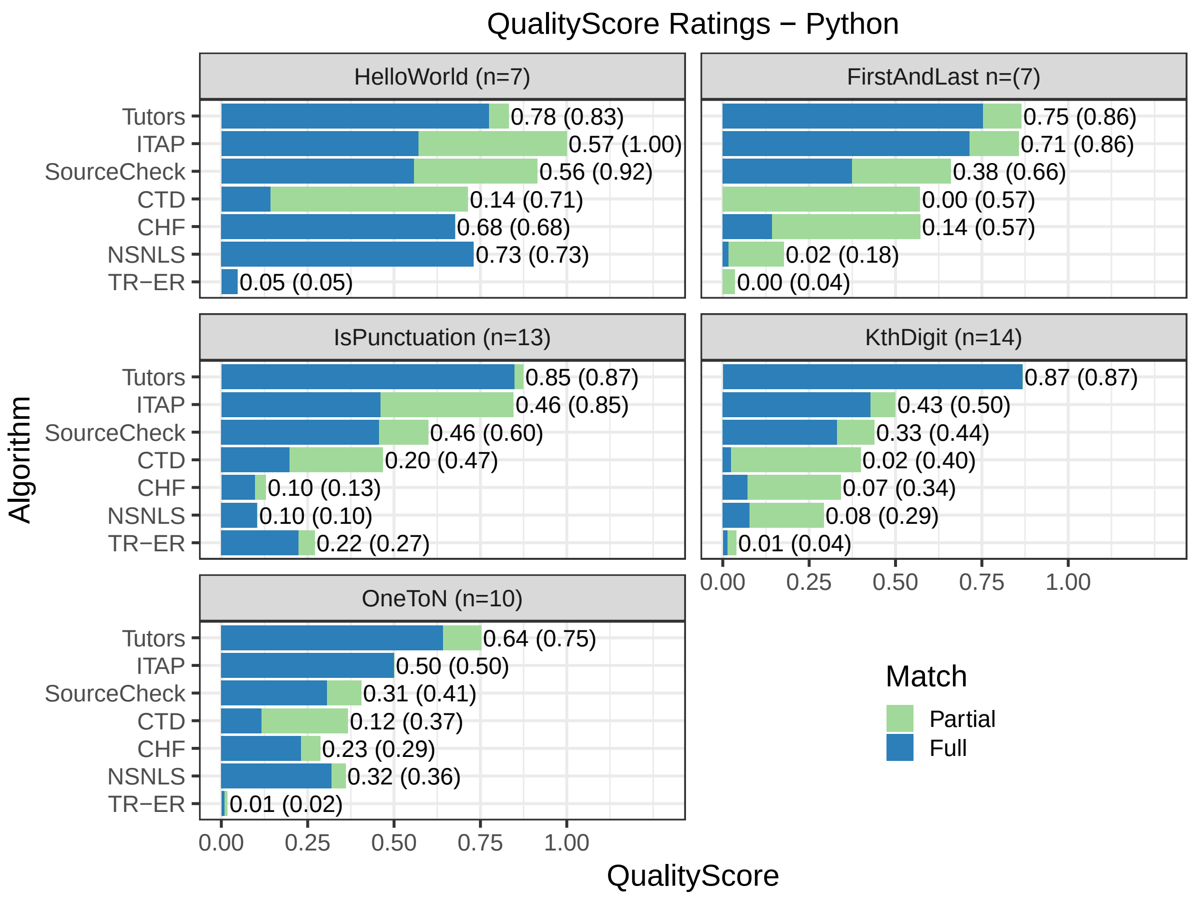


Fig.5. QUALITYSCORE ratings for each algorithm on each assignment in the Python dataset.

that no binary factor separates "good" and "bad" algorithms. The performance ranks of the algorithms are also consistent across problems and datasets. All algorithms achieved the same ranks on both problems in the iSnap dataset (see Figure 4), which also matches the overall rankings in the Python dataset (counting partial matches). There is more variance among the rankings on the Python problems (see Figure 5), but the top-two performing algorithms are consistently ITAP and SourceCheck, respectively.

There are meaningful differences in the algorithms' scores and ranks, depending on whether we include partial matches in the QUALITYSCORE, especially on the Python dataset. The CTD algorithm provides the clearest example, since its QUALITYSCORE on the Python dataset rises from 0.10 (rank 5) to 0.48 (rank 3) if partial matches are included. Recall that a hint *partially matches* a gold standard hint if it consists of a subset of the edits suggested by a gold standard hint, and some of these edits add code. Table 2 gives examples of partial matches generated by CTD on each problem. Partial matches often occurred when a hint inserted new, correct code that failed to incorporate the student's existing, correct

Table 1
Detailed performance data for each algorithm on both datasets. The columns respectively give the QualityScore (using full and partial hint matching), the percentage of hint requests for which the algorithm produced any valid hints, the average number of *valid* hints generated per hint request, and the average number of *total* hints generated per hint request.

Dataset	**Algorithm**	**QualityScore**		**% Requests w/ Any Valid**		**Avg. # Valid Hints**		**Avg. # Hints**
		Full	**Partial**	**Full**	**Partial**	**Full**	**Partial**	
iSnap	SourceCheck	0.38	0.44	0.79	0.82	1.46	1.75	6.03
	CTD	0.24	0.37	0.67	0.82	1.07	1.71	5.13
	CHF	0.15	0.17	0.64	0.74	1.15	1.33	8.84
	NSNLS	0.13	0.14	0.56	0.57	0.72	0.80	5.79
	TR-ER	0.04	0.05	0.38	0.41	0.57	0.67	15.46
Python	ITAP	0.51	0.71	0.51	0.71	0.51	0.71	1.00
	SourceCheck	0.40	0.57	0.53	0.77	0.59	0.94	2.20
	CTD	0.10	0.48	0.20	0.65	0.22	0.86	2.06
	CHF	0.20	0.35	0.35	0.57	0.51	0.88	3.18
	NSNLS	0.21	0.30	0.37	0.49	0.39	0.53	2.20
	TR-ER	0.07	0.09	0.18	0.31	0.18	0.33	8.12

code (e.g., rows 1, 2, 3 and 6). On Python problems, the specifics of the AST caused issues, for example when a hint would suggest adding a binary operator node (`BinOp`), without adding an additional node to specify *which* operator (e.g., rows 4 and 6). Often, partial matches simply provided less specific information (e.g., row 2, 3, 4, 6, 7), but rows 1 and 5 show how they can also fail to remove erroneous code. Partial matches represent a variety of hint types, each containing possibly useful information but also falling short of a high-quality hint. These hints may still be useful to students, especially those with more ability to self-explain the hint, but they are clearly distinct from full matches.

Table 2
Examples of partial matches generated by CTD on each problem

Problem	**Original Code**	**Gold Standard Hint**	**Partial Match Hint**

1. GuessingGame	`number = answer`
	`doUntil(answer == random(1, 10))`
	`{...}`
		`number = random(1, 10)`
		`doUntil(answer == number)`
		`{...}`
			`number = random(1, 10)`
			`doUntil(answer == random(1, 10)`
			`{...}`

2. Squiral	`repeat(x) {...}`	`repeat(x * __) {...}`	`repeat(__ * __) {...}`
3. HelloWorld	`'Hello World'`	`return 'Hello World'`	`return __`
4. FirstAndLast	`return`	`return __ + __`	`return __ BinOp __`
5. IsPunctuation	`return str1.punctuation == '!'`	`return __ in string.__`	`return str1.punctuation in '!'`
6. KthDigit	`return (x//10**k)%10`	`return (x//10**(k-__))%10`	`return (x//10**(__ BinOp __))%10`
7. OneToN	`str1[i] = i`	`str1 += __`	`__ += __`

RQ3 asks how state-of-the-art algorithms compare to humans for the task of next-step hint generation. On the iSnap dataset, the best algorithm (SourceCheck) performed about half as well as human tutors, while on the Python dataset, the best algorithm (ITAP) performed up to 84% as well (including partial matches), and the difference between them was not significant. This suggests that for the simpler Python problems, a data-driven algorithm can approach the quality and consistency of human-authored, next-step hints. This is very impressive, especially considering how new the research area is. However, this does not mean that access to data-driven hints will always improve student learning. Previous work has shown that many students will not request on-demand hints, even when they would benefit from them (Aleven et al. 2003; Price et al. 2017c; Roll et al. 2011). Additionally, even *human-quality* next-step programming hints are still effectively "bottom-out" hints, which may not be the most effective form of help (Aleven et al. 2016), since students may need to self-explain the hint in order to learn from it (Shih et al. 2008). Still, these results strongly suggest the need for controlled, classroom studies on data-driven, next-step hints to accurately measure their impact on student learning. For the more complex problems in the iSnap dataset, data-driven hints still fall well short of human-authored hints *on average*. However, we argue that they still succeed in many situations, and in the next section we explore how we can better identify these situations.

CHALLENGES FOR DATA-DRIVEN HINT GENERATION

RQ4 asks what factors contribute to poor-quality hints and how we can improve current data-driven hint generation algorithms. To answer this, we manually investigated hint requests in both the iSnap and Python datasets where the hint generation algorithms generally performed poorly. We created a simple metric for the difficulty of a given hint request, defined as 1 minus the *second-highest* QUALITYSCORE achieved by any algorithm on that request, calculated using partial matches. We used the second-highest QUALITYSCORE (80th percentile), rather than median (50th percentile) because we were most interested in challenges faced by the best-performing algorithms. We manually inspected 51 hint requests that had a difficulty greater than 0.75, meaning that at most one algorithm performed even remotely well, with a QUALITYSCORE of 0.25 or higher. Based on this investigation, along with our results from Experiment 1, we developed several hypotheses about what factors may make data-driven hint generation difficult, which are presented in this section. We also investigated whether the data supported these hypotheses, analyzing all 112 hint requests. To investigate these hypotheses across algorithms, we also define the difficulty of a hint request for any given algorithm as 1 minus the algorithm's QUALITYSCORE on that request (including partial matches).

Difficulty from the Hint Requests

We hypothesized that some difficulties may be inherent in the hint requests themselves. There were significant differences in all algorithms' QUALITYSCORE ratings between different hint requests on both the iSnap (Kruskal-Wallis test; $H(60) = 99.9$; $p < 0.001$) and Python ($H(50) = 108.4$; $p < 0.001$) datasets, which supports this hypothesis. We investigated the following hypotheses about which attributes of a hint request make it difficult:

Large Code: We hypothesized that when a student has written a lot of code, hint generation algorithms may have difficulty identifying the key areas where a student needs help, leading to lower-quality hints. This hypothesis was supported, as we found a positive, significant Spearman's correlation between the size of a hint request's AST and the difficulty of that hint request for both the iSnap ($r_s = 0.376$; $p = 0.003$) and Python ($r_s = 0.389$; $p = 0.005$) datasets. As shown in the third column of Table 3, this correlation is positive for each algorithm on each dataset, suggesting that AST size makes hint genera-

Table 3
Correlations between each of the variables we hypothesized may contribute to hint generation difficulty and the *difficulty* metric (1 - QUALITYSCORE) for each algorithm on each dataset, calculated across all *hint requests*. The final column gives the comparative likelihood of a deletion hint being invalid (CLI) compared to a non-deletion hint, calculated across all *hints generated*. Empty cells indicate no variance within the algorithm (e.g., ITAP always produce 1 hint and no deletions).

Algorithm	Spearman's Correlation between Variable and Difficulty					**Deletion**
	Dataset	**Size**	**Divergence**	**# GS Hints**	**# Alg Hints**	**CLI**
TR-ER	iSnap	0.357	0.142	-0.096	0.169	7.662
TR-ER	Python	0.309	0.140	-0.029	0.460	4.129
NSNLS	iSnap	0.220	0.137	-0.241	0.178	4.431
NSNLS	Python	0.262	0.171	0.099	0.427	-0.463
CHF	iSnap	0.100	0.164	-0.155	0.179	2.600
CHF	Python	0.118	0.333	0.042	0.244	1.265
CTD	iSnap	0.428	0.461	-0.272	0.275	14.963
CTD	Python	0.163	0.070	-0.223	0.220	2.967
SourceCheck	iSnap	0.094	0.177	0.068	0.535	
SourceCheck	Python	0.347	0.346	0.031	0.675	
ITAP	Python	0.059	0.196	-0.134		

tion difficult for all algorithms. One reason for algorithms' difficulty with larger hint requests may be a tendency to generate *too many* hints, as explored in the next section.

Divergent Code: We hypothesized that *divergent* hint requests, which feature code that is unlike most other student solutions, would be more difficult for hint generation. For example, on the GuessingGame, one student started by asking, "Do you want to play the game?" and put their remaining code inside of a conditional statement, requiring the player to say "yes." This was a unique design choice, and none of the six algorithms generated a valid hint for this request. To investigate this hypothesis, we defined the *divergence* of a given hint request as the median tree edit distance between that hint request's code and each solution in the training dataset for that problem. We found a significant, positive correlation between the divergence of a hint request and its difficulty in the iSnap ($r_s = 0.356$; $p = 0.005$) and itap ($r_s = 0.432$; $p = 0.002$) datasets. As shown in the fourth column of Table 3, this correlation is positive for each algorithm on each dataset, suggesting it is a consistent trend. This supports our hypothesis that divergent code is more difficult for the algorithms. However, we also found a significant correlation between the hint request's divergence and AST size (as discussed above) for the iSnap ($r_s = 0.385$; $p = 0.002$) and Python ($r_s = 0.458$; $p = 0.001$) datasets, so these effects may be related.

Few Correct Hints: The number of valid, gold standard hints varied considerably among hint requests, from 1 to 11 in the iSnap dataset (Med = 5) and 1 to 6 in the Python dataset (Med = 2). We hypothesized that hint requests with fewer gold standard hints would be more difficult for hint algorithms, as there were fewer possible ways to produce a valid hint. However, the data did not support this hypothesis, as there was no significant Spearman's correlation between the difficulty of a hint request and the number of gold standard hints generated by tutors for the iSnap ($r_s = -0.225$; $p = 0.081$) or Python ($r_s = 0.072$; $p = 0.614$) datasets. As shown in the fifth column of Table 3, this correlation had no consistent trend across algorithms.This may be because those hint requests which elicited fewer gold standard hints were also more straightforward, reducing difficulty.

Difficulty from the Algorithms

While some hint requests proved more difficult than others, there was also a clear difference in quality between algorithms. In this section, we investigate hypotheses for why some algorithms achieved lower QUALITYSCORES. We report all QUALITYSCORES calculated *with* partial matches, but the general trends hold whether or not partial matches are included. Part of our analysis is informed by Experiment 1, where tutors were asked to rate the quality of the algorithms' hints. When they rated a hint as invalid, they also supplied a reason: Incorrect, Too Much Information, Unhelpful, Too Soon, or Impossible.

Unfiltered Hints: The average number of hints generated per hint request ranged from 5.1 to 15.5 for algorithms on the iSnap dataset and from 1 to 8.1 on the Python dataset (the ITAP algorithm always generated 1 hint). We hypothesized that an algorithm generating a large number of hints might indicate a failure to identify the most important hints, leading to lower-quality hints overall. This hypothesis was supported by the data, as there was a significant, negative Spearman's correlation between the number of hints an algorithm generated for a given hint request and its difficulty on that request for both the iSnap ($r_s = 0.437$; $p < 0.001$) and Python ($r_s = 0.487$; $p < 0.001$) datasets. As shown in the sixth column of Table 3, this correlation was also positive for each algorithm individually on both datasets (with the exception of the ITAP algorithm, which always produced 1 hint). This result is not obvious, since generating multiple hints with appropriate confidence weights could result in more consistent (but equally good) performance than attempting to select a single, best hint.

Algorithms generated more hints for larger hint request ASTs in both the iSnap ($r_s = 0.307$; $p = 0.016$) and Python ($r_s = 0.487$; $p < 0.001$) datasets. However, there was *no* significant Spearman's correlation between the size of a hint request's AST and the number of hints the *human tutors* authored for the iSnap ($r_s = -0.222$; $p = 0.083$) or Python ($r_s = 0.189$; $p = 0.184$) datasets. There was also no significant correlation between size and number of hints for SourceCheck on the iSnap dataset and ITAP on the Python dataset (since it always generates 1 hint). This suggests that, for our human tutors and our top-performing algorithms, the number of hints generated was not strongly impacted by the amount of code in the hint request.

Incorrect and Unhelpful Deletions: We noticed that many of the hints that we manually rated as invalid in Experiment 1 were deletions, with 40% of those marked as Unhelpful and 47% as Incorrect. We hypothesized that it may be difficult for algorithms to distinguish between deleting code that is incorrect, and code that is simply unnecessary but also not harmful. For the iSnap and Python datasets respectively, 17.7% and 18.0% of hints generated by all algorithms were deletions (removed code without adding any). However, only 17 deletion hints (2.8% of deletions) matched the gold standard across all algorithms and datasets, and of these, all but 5 were duplicate hints produced by different algorithms. Notably, the two best performing algorithms, SourceCheck and ITAP, did not produce deletion hints. A primary reason that deletions were poorly rated is that only 3.2% and 5.1% of gold standard hints were deletions for the iSnap and Python datasets, respectively. However, while approximately half of gold standard hints were matched to some algorithmic hint, only 25% of gold standard *deletion* hints were matched. As shown in the final column of Table 3, deletion hints were more likely to be rated *invalid* than non-deletion hints for each algorithm and dataset, with one exception (NSNLS on Python). This suggests that useful deletion hints were rare and the algorithms consistently failed to produce them.

Of the 5 unique deletion hints that matched the gold standard, one removed a `break` statement that erroneously ended a loop, two removed a variable declaration that masked a function parameter, and two removed an unneeded function call. In each case, the deleted code was problematic on its own, and did not need to be replaced with other code. Of the over 600 deletions that did not match the gold standard, many deleted code that was unnecessary but not harmful, or suggested removing erroneous code (sometimes large sections of it) without a meaningful replacement.

Understanding Student Intent: In Experiment 1, when our human tutors rated hints, they often discussed the hint-requesting student's intent, referring to their identifier names and code history to determine intent. We hypothesized that algorithms would struggle with this aspect of hint generation, since they do not reason about natural language and only the CHF reasoned about a student's code history. While this hypothesis is difficult to test quantitatively, we did find some evidence in the data. For example, on the Squiral assignment, students had to use one variable (length), and one function parameter (rotations). However, when students instead named their *function parameter* "length" (or "move," "size," etc.), the human tutors would compensate by adjusting their hints to use this parameter instead of a length *variable*. The algorithms were rarely able to do so. Our tutors were also able to distinguish between meaningful and arbitrary literal values. For example on Squiral, the number of sides per rotation *must* be 4, but the size of the shape can vary. This led to a number of false positives and false negatives for the hint generation algorithms that changed or failed to change literal values. This was more common on the iSnap dataset, where more implementation choices in assignments were left up to the students.

Probably the most important way that tutors recognized student intent was by focusing their hints on the most relevant problems facing the student. Their hints focused on code recently edited by the student, code with errors that would block forward progress, and code that reflected important misunderstandings. By contrast, many algorithms suggested hints *anywhere* progress could be made. Sometimes these hints were reasonable, but came *too soon*, suggesting fixes *before* a student was ready to process that information. Of the hints marked invalid in Experiment 1, 13% were labeled "Too Soon." For example, a student who has initialized a variable incorrectly, reflecting a clear misunderstanding of its purpose, may be confused by a hint that suggests where to use that variable until the error is corrected. This is related to the challenge of producing *too many* hints, but it can still occur with just one hint.

Discussion – RQ4

Our results identify key areas where data-driven hint generation algorithms generally fail to produce high-quality hints. We can use this information in two key ways. First, we can ensure that future work addresses the current weaknesses in data-driven hints generation algorithms. One area where hint algorithms can clearly improve is filtering and selecting hints. This has been identified in previous work (Price et al. 2017d) as a primary reason data-driven hints fail to match the quality of human-authored ones. It is notable that the highest-performing algorithm, ITAP, generates only one hint per request. ITAP uses a customized desirability metric to first select the most useful macro-level edit, and then it creates a token-level hint by selecting the edit closest to the root node of the AST (Rivers and Koedinger 2017). Other algorithms might benefit by adopting a similar approach. Another clear area for improvement is deletion hints, which were rarely valid. The simplest solution is to avoid deletions unless they suggest replacement code, as ITAP and SourceCheck do. When used in the iSnap system, SourceCheck does generate deletion hints, but it does so with a passive warning highlight, rather than as a next-step hint. Lastly, many algorithms do not use the *history* of a student's code, prior to the hint request, to inform hint generation and target relevant and recently-edited code. While ITAP, CHF and CTD use the history of traces in the *training dataset*, only the CHF algorithm utilizes the history of the hint-requester. By addressing these challenges, we should be able to produce higher-quality data-driven hints.

However, some barriers we identified, such as understanding student intent, may not be easy to overcome. A second way we can use the findings presented here is to identify hint requests where automated hints are likely to fail, so that algorithms can avoid giving low-quality hints that may erode trust in the system (Price et al. 2017c). Instead, a hint system could recommend that the student seek out additional help from an instructor or peer. For example, it is straightforward to recognize divergent code in hint requests by comparing it to other solutions in the training dataset. For these hint requests, we found that algorithms produced lower-quality hints, so they may do more good by *not* giving hints. However, it is also important to remember that data-driven hint generation for programming is still a relatively new research area; these shortcomings emphasize just how challenging the task is.

CONCLUSION

In this work, we have presented the following primary contributions: 1) the validated QUALITYSCORE procedure for evaluating and comparing the quality of next-step programming hints, 2) the first comparison of hint generation algorithms across multiple datasets, 3) insights into the current strengths and limitations of data-driven hints for programming, and 4) publicly available datasets that we encourage other researchers to use to evaluate, compare and refine hint generation algorithms. Specifically, we found that the QUALITYSCORE procedure produces hint ratings that are consistent and replicable, which agree with manual tutor ratings. We used the procedure to evaluate six algorithms on two novice programming datasets, with different programming languages, finding large differences in hint quality that were fairly consistent across datasets. The SourceCheck and ITAP algorithms performed significantly better than the other algorithms, and on the simpler problems of the Python dataset, ITAP generates near-human-quality hints. Despite these successes, we also identified situations where data-driven hints perform poorly overall, such as when students have code that diverges from common solutions. Currently, algorithms struggle to distinguish between important and unimportant deletions and fail to prioritize the most relevant hints. In addition to these insights gained from comparing six specific data-driven hint generation algorithms, we have also provided a method and datasets that will allow others to perform QUALITYSCORE evaluations on additional algorithms, enabling a more rigorous approach to evaluating data-driven hints.

This work has important limitations. The QUALITYSCORE procedure itself attempts to operationalize the notion of hint quality, but like any operationalization, it does not capture all of the relevant aspects. The procedure rates hints on a binary scale as valid or invalid, and it has no easy way to address marginal-quality hints, or the differences between "good" and "great" hints. Our procedure also does not consider how *many* of the gold standard hints an algorithm generates, so a set of 3 valid hints and 1 invalid hint would currently earn a lower QualityScore (0.75) that a set with 1 valid hint (1.0). This intentional choice was because most hint interfaces display just one hint at a time, so generating multiple useful hints per hint request may not be as important as avoiding invalid hints. Our gold standard hints came from only 3 experts tutors per dataset, and a different set of expert raters might produce different gold standards. Further, the QualityScore is based on expert opinions, which may not necessarily correspond to more important measures of quality, such as programming outcomes and learning, or students' trust in and perceived value of hints. However, there are few objective measures of the quality of intelligent support for programmers, and the QualityScore provides a way to benchmark their performance.

In future work, we hope to explore more nuanced ways to rate generated hints (e.g., in Paaßen et al. 2018), beyond the current binary valid/invalid rating used by QualityScore, especially for partially valid hints. Future work should also evaluate the quality of other forms of data-driven feedback for programming, such as example-based help (Gross et al. 2014b), error flagging hints (Lazar et al. 2017) and style-focused hints (Choudhury et al. 2016; Moghadam et al. 2015), as well as in other domains such as logic proofs (Barnes and Stamper 2008). Most importantly, the results from this evaluation suggest a strong need to evaluate data-driven programming hints in learning settings with students.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under grant 1623470.

*

REFERENCES

V. Aleven, E. Stahl, S. Schworm, F. Fischer, and R. Wallace. Help Seeking and Help Design in Interactive Learning Environments Vincent. *Review of Educational Research*, 73(3):277–320, 2003.

V. Aleven, I. Roll, B. M. McLaren, and K. R. Koedinger. Help Helps, But Only So Much: Research on Help Seeking with Intelligent Tutoring Systems. *International Journal of Artificial Intelligence in Education*, 26 (1):1–19, 2016. ISSN 1560-4292. doi: 10.1007/s40593-015-0089-1.

N. Augsten, M. Böhlen, and J. Gamper. Approximate matching of hierarchical data using pq-grams. *21st Int. Conf. on Very large data bases (VLDB)*, pages 301–312, 2005.

T. Barnes and J. Stamper. Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 373–382, 2008.

T. Barnes and J. Stamper. Automatic hint generation for logic proof tutoring using historical data. *Educational Technology & Society*, 13(1):3–12, 2010. ISSN 11763647. doi: 10.1007/978-3-540-69132-7-41.

Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society*, 57(1):289–300, 1995. ISSN 00359246. doi: 10.2307/2346101.

R. R. Choudhury, H. Yin, and A. Fox. Scale-driven automatic hint generation for coding style. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 122–132, 2016. ISBN 9783319395821. doi: 10.1007/978-3-319-39583-8_12.

S. Chow, K. Yacef, I. Koprinska, and J. Curran. Automated Data-Driven Hints for Computer Programming Students. In *Adjunct Publication of the Conference on User Modeling, Adaptation and Personalization*, pages 5–10, 2017. ISBN 978-1-4503-5067-9. doi: 10.1145/3099023.3099065.

A. Corbett and J. R. Anderson. Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes. In *Proceedings of the SIGCHI Conference on Human Computer Interaction*, pages 245–252, 2001.

D. Fossati, B. Di Eugenio, S. Ohlsson, C. Brown, and L. Chen. Data Driven Automatic Feedback Generation in the iList Intelligent Tutoring System. *Technology, Instruction, Cognition and Learning*, 10(1):5–26, 2015.

S. Gross, B. Mokbel, B. Hammer, and N. Pinkwart. How to Select an Example? A Comparison of Selection Strategies in Example-Based Learning. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 340–347, 2014a.

S. Gross, B. Mokbel, B. Paassen, B. Hammer, and N. Pinkwart. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology*, 9(3):248, 2014b. ISSN 1477-8386. doi: 10.1504/IJLT.2014.065752.

R. Gupta, S. Pal, A. Kanade, and S. Shevade. DeepFix: Fixing Common Programming Errors by Deep Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 1, pages 1345–1351, 2017.

B. Hartmann, D. Macdougall, J. Brandt, and S. R. Klemmer. What Would Other Programmers Do? Suggesting Solutions to Error Messages. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 1019–1028, 2010. ISBN 9781605589299. doi: 10.1145/1753326.1753478.

A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the ACM Conference on Learning @ Scale*, pages 89–98, 2017. ISBN 9781450344500. doi: 10.1145/3051457.3051467.

W. Jin, T. Barnes, and J. Stamper. Program Representation for Automatic Hint Generation for a Data-driven Novice Programming Tutor. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 1–6, 2012.

K. Koedinger and J. Stamper. Using data-driven discovery of better student models to improve student learning. In *Proceedings of the International Conference on Artificial Intelligence in Education*, 2013.

K. R. Koedinger, R. S. J. Baker, K. Cunningham, and A. Skogsholm. A Data Repository for the EDM community: The PSLC DataShop. In C. Romero, S. Ventura, M. Pechenizkiy, and R. S. Baker, editors, *Handbook of Educational Data Mining*, pages 43–55. CRC Press, 2010. ISBN 1439804583. doi: doi:10.1201/b10274-6.

T. Lazar and I. Bratko. Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 306–311. Springer, 2014.

T. Lazar, M. Možina, and I. Bratko. Automatic Extraction of AST Patterns for Debugging Student Programs. In *Proceedings of the International Conference on Artificial Intelligence in Education*, pages 162–174, 2017. ISBN 978-3-319-61425-0. doi: 10.1007/978-3-319-61425-0_14.

J. B. Moghadam, R. R. Choudhury, H. Yin, and A. Fox. AutoStyle: Toward Coding Style Feedback At Scale. In *Proceedings of the ACM Conference on Learning @ Scale*, pages 261–266, 2015. ISBN 9781450334112. doi: 10.1145/2724660.2728672.

B. Mostafavi, G. Zhou, C. Lynch, M. Chi, and T. Barnes. Data-driven worked examples improve retention and completion in a logic tutor. In *Proceedings of the International Conference on Artificial Intelligence in Education*, volume 9112, pages 726–729, 2015. ISBN 9783319197722. doi: 10.1007/978-3-319-19773-9_102.

B. Paaßen, C. Göpfert, and B. Hammer. Time Series Prediction for Graphs in Kernel and Dissimilarity Spaces. *Neural Processing Letters*, pages 1–21, 2017. ISSN 1573773X. doi: 10.1007/s11063-017-9684-5.

B. Paaßen, B. Hammer, T. W. Price, T. Barnes, S. Gross, and N. Pinkwart. The Continuous Hint Factory -Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *Journal of Educational Data Mining*, pages 1–50, 2018.

B. Peddycord III, A. Hicks, and T. Barnes. Generating Hints for Programming Problems Using Intermediate Output. In *Proceedings of the International Conference on Educational Data Mining*, pages 92–98, 2014.

D. Perelman, S. Gulwani, and D. Grossman. Test-Driven Synthesis for Automated Feedback for Introductory Computer Science Assignments. In *Proceedings of the Workshop on Data Mining for Educational Assessment and Feedback*, 2014.

C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *Proceedings of the International Conference on Machine Learning*, pages 1093–1102, 2015a. ISBN 9781510810587 (ISBN).

C. Piech, M. Sahami, J. Huang, and L. Guibas. Autonomously Generating Hints by Inferring Problem Solving Policies. In *Proceedings of the ACM Conference on Learning @ Scale*, pages 1–10, 2015b.

T. Price, R. Zhi, Y. Dong, N. Lytle, and T. Barnes. The impact of data quantity and source on the quality of data-driven hints for programming. In *Proceedings of the International Conference on Artificial Intelligence in Education*, 2018. ISBN 9783319938424. doi: 10.1007/978-3-319-93843-1_35.

T. W. Price, Y. Dong, and T. Barnes. Generating Data-driven Hints for Open-ended Programming. In *Proceedings of the International Conference on Educational Data Mining*, 2016.

T. W. Price, Y. Dong, and D. Lipovac. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2017a. ISBN 9781450346986.

T. W. Price, Z. Liu, V. Catete, and T. Barnes. Factors Influencing Students' Help-Seeking Behavior while Programming with Human and Computer Tutors. In *Proceedings of the International Computing Education Research Conference*, 2017b. ISBN 9781450349680.

T. W. Price, R. Zhi, and T. Barnes. Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior. In *Proceedings of the International Conference on Artificial Intelligence in Education*, 2017c.

T. W. Price, R. Zhi, and T. Barnes. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. In *Proceedings of the International Conference on Educational Data Mining*, 2017d.

Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay. sk_p: a neural program corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 39–40. ACM, 2016.

K. Rivers and K. Koedinger. Automatic Generation of Programming Feedback: A Data-driven Approach. In *Proceedings of the First Workshop on AI-supported Education for Computer Science*, pages 50–59, 2013.

K. Rivers and K. Koedinger. Automating Hint Generation with Solution Space Path Construction. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 329–339, 2014.

K. Rivers and K. R. Koedinger. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64, 2017.

K. Rivers, E. Harpstead, and K. Koedinger. Learning Curve Analysis for Programming: Which Concepts do Students Struggle With? In *Proceedings of the International Computing Education Research Conference*, pages 143–151, 2016. ISBN 9781450344494.

I. Roll, V. Aleven, B. M. McLaren, and K. R. Koedinger. Improving Students' Help-seeking Skills Using Metacognitive Feedback in an Intelligent Tutoring System. *Learning and Instruction*, 21(2):267–280, 2011. ISSN 09594752. doi: 10.1016/j.learninstruc.2010.07.004. URL `http://dx.doi.org/10.1016/j.learninstruc.2010.07.004`.

B. Shih, K. Koedinger, and R. Scheines. A Response Time Model for Bottom-Out Hints as Worked Examples. In *Proceedings of the International Conference on Educational Data Mining*, pages 117 – 126, 2008. ISBN 9780615306292. doi: doi:10.1201/b10274-17.

R. Suzuki, G. Soares, A. Head, and E. Glassman. TraceDiff : Debugging Unexpected Code Behavior Using Trace Divergences. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, 2017. ISBN 9781538604434.

K. VanLehn. The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education*, 16 (3):227–265, 2006.

T. K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):81–88, 1968. ISSN 0011-4235, 1573-8337. doi: 10.1007/BF01074755.

K. Wang, B. Lin, B. Rettig, P. Pardi, and R. Singh. Data-Driven Feedback Generator for Online Programing Courses. In *Proceedings of the ACM Conference on Learning @ Scale*, pages 257–260, 2017. ISBN 9781450344500. doi: 10.1145/3051457.3053999.

C. Watson, F. W. B. Li, and J. L. Godwin. BlueFix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *Proceedings of the International Conference on Web-based Learning*, pages 228–239, 2012. ISBN 9783642336416. doi: 10.1007/978-3-642-33642-3_25.

J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 740–751, 2017. ISBN 9781450351058. doi: 10.1145/3106237.3106262.

M. Yudelson, R. Hosseini, A. Vihavainen, and P. Brusilovsky. Investigating Automated Student Modeling in a Java MOOC. In *Proceedings of the International Conference on Educational Data Mining*, pages 261–264, 2014.

K. Zhang and D. Shasha. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989. ISSN 0097-5397. doi: 10.1137/0218082.

K. Zimmerman and C. R. Rupakheti. An Automated Framework for Recommending Program Elements to Novices. In *Proceedings of the International Conference on Automated Software Engineering*, 2015. ISBN 9781509000258. doi: 10.1109/ASE.2015.54.