Identifying typical approaches and errors in Prolog programming with argument-based machine learning

Martin Možina[a,*], Timotej Lazar[a], Ivan Bratko[a]

[a] *Faculty of Computer and Information Science, University of Ljubljana, Večna pot 113, 1000 Ljubljana, Slovenia*

Abstract

Students learn programming much faster when they receive feedback. However, in programming courses with high student-teacher ratios, it is practically impossible to provide feedback to all homeworks submitted by students. In this paper, we propose a data-driven tool for semi-automatic identification of typical approaches and errors in student solutions. Having a list of frequent errors, a teacher can prepare common feedback to all students that explains the difficult concepts. We present the problem as supervised rule learning, where each rule corresponds to a specific approach or error. We use correct and incorrect submitted programs as the learning examples, where patterns in abstract syntax trees are used as attributes. As the space of all possible patterns is immense, we needed the help of experts to select relevant patterns. To elicit knowledge from the experts, we used the argument-based machine learning (ABML) method, in which an expert and ABML interactively exchange arguments until the model is good enough. We provide a step-by-step demonstration of the ABML process, present examples of ABML questions and corresponding expert's answers, and interpret some of the induced rules. The evaluation on 42 Prolog exercises further shows the usefulness of the knowledge elicitation process, as the models constructed using ABML achieve significantly better accuracy than the models learned from human-defined patterns or from automatically extracted patterns.

*Corresponding author. Tel.:+38614798292

Email addresses: martin.mozina@fri.uni-lj.si (Martin Možina), timotej.lazar@fri.uni-lj.si (Timotej Lazar), ivan.bratko@fri.uni-lj.si (Ivan Bratko)

Keywords: argument-based machine learning, rule learning, programming tutors, abstract syntax tree, syntactic patterns

1. Introduction

Programming is nowadays a must-have skill for professionals in many disciplines, and is becoming a part of basic education in many countries. It is being taught at many universities and in massive open online courses (MOOCs). Learning programming requires a great deal of practice. Bloom (1984) has shown that students learn much faster when a tutor is available to help select exercises, explain errors, and suggest possible problem-solving steps. However, due to high student-teacher ratios at universities – and even more so in MOOCs – it is practically impossible for human tutors to evaluate each individual work.

One solution is to use an intelligent tutoring system, where students implement programs in a controlled environment and receive help whenever needed. Such systems can be, in some cases, as effective as a human tutor (VanLehn, 2011). The alternative is to let students use their preferred environment, solve the problem by themselves, submit solutions and then receive automatically generated feedback. Both approaches, however, rely on a difficult knowledge acquisition task: we need to encode the teacher's knowledge in an explicit form.

To address this problem we propose a machine learning algorithm for learning rules to distinguish between correct and incorrect programs. These rules describe different approaches that students used to solve exercises. A rule describing an incorrect program contains explicit reasons why this program is incorrect. Conversely, a rule describing correct programs specifies necessary components for a program to be correct. The semantics of these rules is analogous to constraint-based tutors (Ohlsson, 1992; Mitrovic, 2012), where constraints specify the necessary properties of correct submissions.

A common problem when using machine learning to build expert systems is that the resulting model is too complex and does not mimic the expert's cognitive processes. It is quite possible to have a system that has high classification

accuracy but is poor at teaching or even explaining its reasoning because a lot of knowledge remains implicit (see (Langley & Simon, 1995) or (Voosen, 2017)).

We encountered a similar problem. To learn from student-submitted programs, which are labeled as "correct" or "incorrect" according to a set of test cases, we decided to use patterns from the programs' abstract syntax trees (ASTs) as attributes for machine learning. However, the space of all possible attributes contains many meaningless AST patterns, which cannot be used for explanation. We therefore needed a programming teacher to specify which AST patterns are useful for distinguishing between correct and incorrect programs. However, the programming teacher was unable to fully express this knowledge in advance.

Domingos (2007) identified several reasons why combining machine learning and expert knowledge often fails, and how it should be approached. One of the reasons is that the results of machine learning are rarely optimal on the first attempt. Iterative improvement, where experts and computer improve the model in turns, is needed. Furthermore, some knowledge is hard to make explicit. It is known that humans are much better at explaining (or arguing) particular cases than explicating general knowledge.

Argument-based machine learning (ABML) (Možina et al., 2007) is an interactive method that helps a domain expert through the mechanism called the ABML knowledge refinement loop. In the loop, the experts are prompted to share only that knowledge which the learning system cannot learn on its own – the experts are asked to provide arguments about selected misclassified examples. Since an argument relates to a single example and the link between its premises and conclusion is presumptive, experts find it easier to explain their knowledge in this way. In our case, instead of asking the expert for all relevant patterns, we ask him or her to provide reasons why a certain program was either correct or incorrect.

This paper presents the following contributions. We first define abstract-syntax-tree paterns and how are they extracted. Then we describe a rule learning algorithm, which bases on the argument-based rule learning algorithm

ABCN2 (Možina et al., 2007), for learning rules that represent typical approaches and errors in student solutions of programming assignments. Afterwards we present an extended version of the ABML refinement loop. In the evaluation section, we show that the patterns, which are the result of applying our algorithm on 42 Prolog exercises from our CodeQ tutoring system[1], lead to accurate machine learned models.

2. Related Work

2.1. Knowledge acquisition for tutoring systems

Domain knowledge for a tutoring system is most often represented with a rule-based model, which is easily understood and modified by a human. Both major ITS paradigms represent domain knowledge with rules: model-tracing tutors use production rules to model the problem-solving process (Anderson et al., 1990; Koedinger & Anderson, 1997), while constraint-based tutors use rules to describe constraints that must hold for every correct solution (Ohlsson, 1992; Mitrovic, 2012).

Creating a human-understandable domain model requires significant knowledge-engineering effort (Murray, 1999; Folsom-Kovarik et al., 2010). This is especially challenging for *programming* tutors because the process of writing a program cannot easily be represented as a sequence of well-defined steps. Furthermore, it is difficult to find meaningful invariant features for comparing programs; most programming exercises have several alternative solutions with many possible implementations (Le et al., 2013).

The first prominent intelligent programming tutor was the LISP tutor (Anderson et al., 1989) with a domain model consisting of over a thousand manually coded production rules describing LISP programming. Constraint-based programming tutors include the J-LATTE Java tutor (Holland et al., 2009)

[1] `https://codeq.si/`

and a tutoring system for Prolog (Le et al., 2009), where domain model was implemented with weighted constraints.

The knowledge-engineering effort required to construct the domain model can be reduced by using machine learning to discover domain knowledge automatically. While machine learning has not yet been used to learn a rule-based domain model for programming, it has been successfully applied to other, more well-defined tutoring domains. Koedinger et al. (Koedinger et al., 2003, 2004), for example, experimented with machine learning to produce production rules from demonstrations of correct and incorrect solutions for a simple math problem. The SimStudent project (Matsuda et al., 2015) explores the same idea: how to effectively learn procedural knowledge from demonstrations provided by teachers and students? Nkambou (Nkambou et al., 2011) combined sequential pattern mining and association rules to learn common patterns of robot arm manipulations recorded by experts. Suraweera et al. (Suraweera et al., 2010) suggested an acquisition system for automatically generating constraints from provided solutions.

Most contemporary programming tutors detect intent and possible misconceptions by comparing the student's program to a set of reference solutions. Suarez and Sison (Suarez & Sison, 2008), for example, implemented a Java tutoring system JavaBugs, which detects the most similar correct program and extracts discrepancies between it and the student's program. Another example is Ask-Elle (Gerdes et al., 2017), which uses reference programs to generate and identify different programming strategies for exercises in Haskell. Various other systems implementing a similar idea are described in (Le, 2016; Keuning et al., 2016).

Writing reference programs is easier than building an explicit domain model. *Data-driven* tutors reduce the necessary effort even further by mining educational data to generate feedback. They construct an implicit domain from solutions submitted by students. In most cases, feedback is still generated from the differences between the student's program and a previously submitted solution.

Rivers & Koedinger (2015) described a data-driven approach for automatic

hint generation. They start by representing the programs as abstract syntax trees (AST), canonicalize them, and then generate a hint for a student from the tree-edit distance between the current student's solution and a reference solution. In our approach, the first two steps are the same. Instead of using a distance measure in the third step, we extract patterns from ASTs and combine them into rules. The usefulness of rule-based knowledge is broader, as it can be used to generate hints or it can be used by a teacher to find out what concepts of the exercise students have problems with. The disadvantage is that our approach can not be fully automated, because the teachers need to define meaningful patterns, as described in Section 5.1.1.

In Codewebs (Nguyen et al., 2014), the authors applied data-driven AST canonicalization with the goal to search for similar submissions more effectively. They argued that standard search engines for code are not appropriate for searching trough students solutions, since the differences between solutions are specific to each exercise. They instead propose to automatically extract "code phrases" from AST, that correspond to different approaches. Because of the large number of all possible patterns, they restrict themselves to subtrees of AST. In this paper we claim that subtrees are not enough to capture student errors and approaches, because many meaningful concepts span over different subtrees. However, similarly as before, generating meaningful patterns across subtrees requires significantly more time from an expert.

In another data driven approach, Jin et al. (2012) represented each program as a linkage graph. In a linkage graph, statements are nodes and edges represents relations between statements. Albeit they use a different representation, they have the same goal as AST-based approaches, as they try to cluster programs with similar linkage graphs and use clusters to generate hints. We believe that it would be possible to combine our approach with their representation, which would result in extraction of meaningful patterns from linkage graphs.

It seems that all data-driven approaches are trying to discover some patterns in programs submitted by students. Whereas other data-driven approaches define a distance between programs and construct feedback based on that, our

algorithm seeks for meaningful and interpretable patterns. Our approach therefore combines ideas from the data-driven methods and ideas from the initial intelligent tutoring systems, where the goal was to produce an explicit (rule-based) model of the domain.

2.2. Structured data mining

Mining programs is closely related to structured data mining, specifically to graph classification, where the task is to classify a graph into one of the predefined classes. One such problem is predicting whether a molecule (represented as a graph) is toxic or not. Our problem definition appears to be the same: predict whether a program (represented as an abstract syntax tree) is correct or not. The most often used models for graph classification are kernel-based and boosting-based methods (Ketkar et al., 2009), however their learned models are difficult to understand and therefore not appropriate for explanation.

On the other hand, frequent subgraph mining aims at finding explainable and interesting patterns in graphs (Jiang et al., 2013). Although this task does not classify graphs, its automatically extracted patterns could be used as attributes in our classification methods. In fact, many approaches exercise this strategy (Bringmann et al., 2011). They first mine frequent subgraphs (patterns) from a given set of learning examples that satisfy some predefined constraints, such as minimal support. Then, optionally, the set of learned subgraphs can be pruned, if the number of unique subgraphs is too large. Afterwards each selected subgraph becomes a binary feature representing whether a learning example contains it or not. The features are then used in a standard machine learning method to build a classifier. In the evaluation section, we compare our semi-automatic technique for pattern extraction with gSpan, a general method for frequent subgraph discovery (Yan & Han, 2002).

2.3. Incremental machine learning

Acquiring domain knowledge is one of the key tasks in machine learning, unfortunately a very difficult one. The problem with domain knowledge occurs

when experts are asked to provide general knowledge, which often fails. On the other hand, asking them to articulate their knowledge iteratively has proved to be much more efficient (Domingos, 2007; Guid et al., 2012).

There are more and more machine learning studies using iterative improvements. Fails & Olsen (2003) used the term *interactive machine learning* to describe an iterative system for correcting errors of an image segmentation system. Since then, researchers have presented many advantages of systems that allow users to interact with machine learning; either to simply obtain new labels, as in active learning, or to improve training supervised classifiers (Amershi et al., 2010). Beside having better final performance, such as accuracy score, these works report that users also gain trust and understanding of their systems.

A particularly interesting interactive approach to machine learning was introduced by Stumpf et al. (2009), where a user can comment on automatically generated explanations provided by a learned model. These comments are then used as constraints in the system when relearning the model. Kulesza et al. (2015) called such an interaction *Explanatory Debugging*, because users identify bugs in a system by inspecting explanations and then explain necessary corrections back to the system.

Explanatory Debugging and argument-based machine learning (ABML) share the same idea: the system starts by explaining the reasons for class value of a certain instance to the user, who in turn can, if the explanation is not satisfactory, provide an alternative reasoning. The system's behavior then needs to change to become consistent with the corrected reasoning. According to the classification of constraint-based machine learning approaches provided by Grossi et al. (2017), ABML and explanatory debugging belong to the same group of algorithms, since they both define constraints at the level of instances, where class value is linked to specific attribute values of an instance.

The main difference between the approaches is how the user is involved in the process. Explanatory Debugging tries to improve the end user's understanding and trust of the model by providing explanations and enabling users to adjust the model's behavior by accepting feedback during regular use of the system.

ABML uses arguments to interact with experts. It tries to minimize the effort that a domain expert needs to exert to build a machine learning model. ABML will ask for explanations of the most difficult examples (see Section 4 for a general description of ABML and Section 4.2 for interactions with experts), because these explanations are supposedly the most informative given the current model. Explanatory Debugging, on the other hand, does not start with the most difficult examples, as one of its goals is to help the users to build a mental model of a working machine learning system. Furthermore, in the existing Explanatory Debugging applications users can not change how data are represented internally, but only how different features influence the system's inference.

3. Dataset

Our educational data consists of Prolog programs submitted by students using the online programming environment CodeQ during the Principles of Programming Languages course at University of Ljubljana. The students start with an empty editor and start programming. When they think that their solution is ready, they submit it for testing. If their program fails the testing cases, they will continue working on until they get it right. We selected 42 exercises with enough submitted programs for machine learning: at least 30 correct submissions and at least 300 submissions in total. Each program was classified as correct if it passed a set of predefined test cases, and incorrect otherwise. Our domain thus covers 42 classification problems with a dichotomous class: correct vs. incorrect.

To use machine learning we must first represent each program in some attribute space. Useful attributes will allow us to distinguish between irrelevant modifications to the program (such as renaming a variable) and modifications that change the meaning of the program (such as changing a function's return value). Such attributes are difficult to define for programming exercises due to high variability of student programs (Nguyen et al., 2014; Piech et al., 2015).

Many different program representations have been used, with either static (based on programs' structure, e.g. (Jin et al., 2012)) or dynamic (based on programs' runtime behavior, e.g. (Glassman et al., 2015)) attributes. Static attributes are easier to handle because the program does not have to be executed, and should still give us enough information to identify errors in a program – teachers can usually find an error without having to execute a program.

We use static attributes defined in terms of abstract-syntax-tree (AST) patterns, which describe parts of the program's structure (Lazar et al., 2017). Tree-based attributes better convey the structure of a program than e.g. using lines of code or other text-based features. Finally, AST patterns are more general than subtrees – as in Codewebs (Nguyen et al., 2014) – because they can encode relations between non-contiguous parts of the program. The remainder of this section describes AST patterns and how they are extracted from student programs.

The idea for AST patterns comes from *tree regular expressions*, mainly used in the field of natural-language processing (Levy & Andrew, 2006). Just as an ordinary regular expression is a string describing a set of strings with a certain structure, an AST pattern is an ordered rooted tree describing a set of ASTs with a certain structure.

We denote AST patterns using s-expressions. For example, the s-expression $(a\ b\ (c\ d))$ denotes a tree with the root a and two children b and c (in that order), where the node c has one child d. Using this notation we define patterns as follows.

The simplest pattern name matches any AST containing a node labeled name. The pattern (name p_1 ... p_k) matches an AST containing a node n labeled name with (at least) k distinct children n_1 to n_k that match, in order, subpatterns p_1 to p_k. Additionally, the operator $*$ can be used to match a chain of zero or more nodes with the same label. For example, the pattern (*a b) matches the trees b, (a b), (a (a y b) z), and so on.

We illustrate AST patterns on the following Prolog program implementing

the relation `sister(X,Y)`[2]:

```
sister(X,Y):-       % X is Y's sister when:
    parent(P,X),
    parent(P,Y),    % X and Y share a common parent P,
    female(X),      % X is female, and
    X \= Y.         % X and Y are not the same person.
```

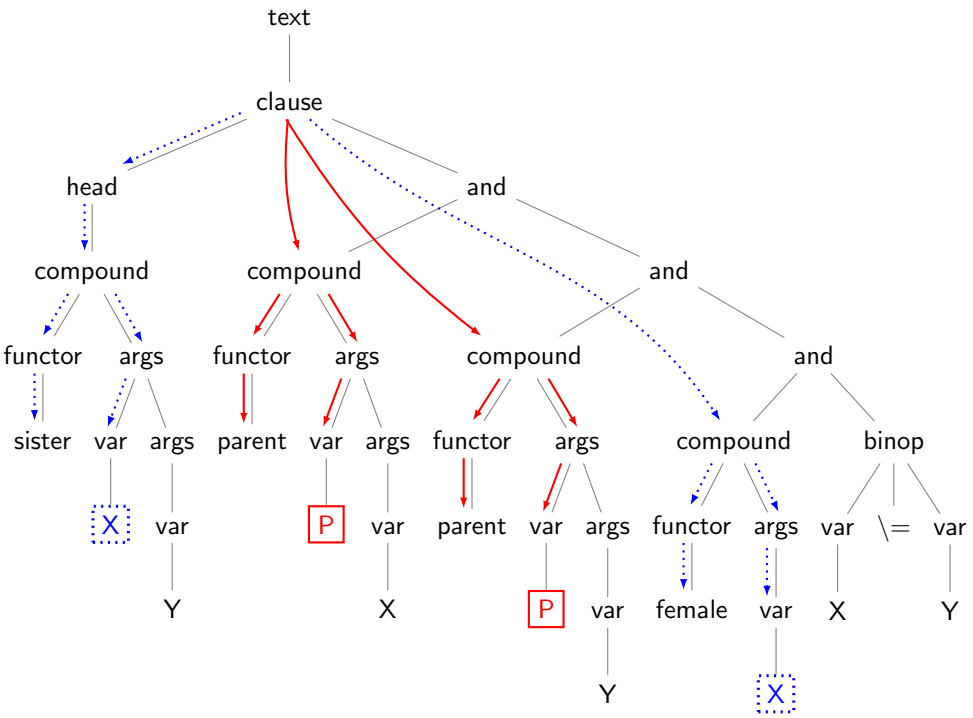

Figure 1: The AST for the `sister` program, showing two patterns and the leaf nodes inducing them. Solid red arrows equate the first arguments in the two calls to `parent`. Dotted blue arrows encode the necessary condition that `X` must be female to be a sister.

Figure 1 shows this program's AST with two patterns. The pattern drawn with blue dotted arrows encodes the fact that the first argument to the `sister` predicate also appears in the call to `female`. In other words, this pattern states that `X` must be female to be a sister. We write this pattern as the s-expression:

[2]Binary relations like this one should be read as "`X` is a sister/parent/... of `Y`".

```
(clause
  (head (compound (functor 'sister') (args var)))
  (*and (compound (functor 'female') (args var))))
```

The $*$ operator in the last line node ensures that the call to `female` may appear anywhere in the clause body (i.e., it may be nested within an arbitrarily deep chain of and nodes). This allows the pattern to match a program regardless of any other goals in the clause.

Since students choose different names for variables, we omit actual variable names from patterns and instead constrain pattern matching to only return matches where all var nodes refer to the same variable. In this study we only consider patterns relating pairs of leaf nodes, because such patterns are easy to interpret. We regard such patterns as the smallest units of meaning in Prolog programs. To model more complex relations, we induce rules that predict correctness based on a combination of patterns.

The second pattern in Fig. 1, drawn with solid red arrows, encodes the fact that the two calls to `parent` share the first argument; in other words, `X` and `Y` must have the same parent `P`. This pattern can be written as:

```
(clause
  (*and (compound (functor 'parent') (args var))
    (*and (compound (functor 'parent') (args var)))))
```

Again, the *and nodes allow the pattern to match any clause with two calls to `parent`. Like the previous example, this pattern relates two instances of the same variable in clause body.

These patterns were extracted automatically from student programs. A pattern is induced for each pair of nodes by traversing the tree upwards and recording all nodes on the path to the clause node. Any sequences of and nodes are replaced with a single *and node in the pattern. For each selected leaf node we also include some local context, such as the predicate name (e.g. `parent`) for compound nodes, or the operator in expressions like `X+1`.

The patterns that occur in at least 5 programs in the learning set form our attribute space. Initially, we constructed patterns where leaf nodes were

referring to the same variable (as in Figure 1). Section 5 demonstrates how argument-based machine learning was used to extend the set of constructed patterns to improve classification accuracy for Prolog programs.

4. Argument-based machine learning

An argument is comprised of a series of premises that are intended to give a reason for the conclusion. Humans mostly use arguments to justify or explain their beliefs and sometimes to convince others. In artificial intelligence, argumentation is a branch that analyzes automatic reasoning from arguments - how arguments for and against a certain claim are produced and evaluated. Argument-based machine learning (ABML) is a combination of argumentation and machine learning.

ABML uses arguments to elicit and represent background knowledge. While providing general background knowledge can be a difficult task for the expert, articulating their knowledge through arguments has proved to be easier. The reason is that in ABML experts need to argue about one specific case at a time and provide knowledge relevant for this case, which does not have to be valid for the whole domain (Domingos, 2007; Guid et al., 2012).

The arguments are retrieved from an expert through the ABML refinement loop, where ABML and the expert exchange arguments in turns. Each provided argument is then attached to a single learning example, while one example can have several arguments. A positive argument is used to explain (or argue) why a certain learning example is in the class as given. In some of the previous applications we also used negative arguments, which speak in favor of the opposite class. However, as negative arguments were not needed in this application, we will, in this paper, refer to positive arguments as arguments. Examples with attached arguments are called *argumented examples*. The dialogue in Section 5.1.1 presents several examples of arguments.

An ABML method needs to induce a model that is consistent with given arguments. That means that the model has to explain the examples using the

same terms that experts used during argumenting these examples. The reasons from the positive argument for a particular example should become a part of the explanation for the class value of this example, and the reasons from the negative arguments should be mentioned within the part speaking against the class value. Such model is therefore more comprehensible from the expert's perspective, since it uses the same terms in explanation as the expert (Guid et al., 2012).

In a medical domain, for example, a machine learning system might explain that a patient has pneumonia, because he is a male and he is coughing. A medical expert could then counter argue, that this person has pneumonia, because he has high temperature. Then, ABML should induce a new model for automatic diagnosis, which would state high temperature (among others) as the reason for this particular patient with pneumonia. However, note that the system does not need to mention temperature in explanations of other examples, in fact, it could even mention low temperature in explanations of other patients with pneumonia, and that would still not violate the constraint.

4.1. Argumented examples and ABCN2

A detailed description of argumented examples and the structure of arguments is given in Mozina et al. (Možina et al., 2007). The following description is a short summary.

Let $(\mathbf{X}, y)$ denote a learning example, where $\mathbf{X}$ is a feature (or attribute) vector and y is a class value. An argumented example is a triple $(\mathbf{X}, y, \mathcal{A})$, where $\mathcal{A}$ is a set containing a) positive arguments explaining why is this example in class y and b) negative arguments explaining why this example should not be in class y. Each argument $a_i \in \mathcal{A}$ is composed of a conjunction of reasons $r_1 \wedge r_2 \wedge \cdots \wedge r_n$. Allowed formats of reasons are:

- $X_i = x_i$ specifies that example has selected class value because (or despite) attribute X_i equals x_i,
- $X_i < x_i$ (or $X_i > x_i$) argues for (or against) class y because the value of

X_i is less than x_i (or is larger),

- $X_i <$ (or $X_i >$) argues for (or against) class y because the value of X_i is low (or high).

ABCN2 is an extended version of the CN2 algorithm (Clark & Boswell, 1991) for learning classification rules from argumented examples. The main difference between the original CN2 and ABCN2 algorithms is in the definition of the covering relation. In the standard definition used in CN2, a rule covers an example if the condition part is true for this example. In ABCN2, a rule R covers an argumented example if the condition part is true and the rule is consistent with the arguments. A rule R AB-covers an argumented example $(\mathbf{X}, y, \mathcal{A})$ if:

- All conditions in R are true for $\mathbf{X}$ (same as in CN2),
- R is consistent with at least one positive argument of $\mathcal{A}$.
- R is not consistent with any of the negative arguments of $\mathcal{A}$.

Using AB-covering by itself leads to learning rules consistent with arguments: rules that contain positive reasons and do not contain negative reasons in their conditions. However, we additionally implemented various changes that made the algorithm more inefficient. For example, in the original CN2 the initial set of candidate rules contains only the default rule, which is then repeatedly specialized until all examples are covered. In ABCN2, we also include the reasons from positive arguments in the initial set of rules, or otherwise the search heuristics might never guide towards rules that contain reasons from positive arguments. The latest version of ABCN2 can be found at `https://github.com/martinmozina/orange3-abml`. All differences between the original ABCN2 described in (Možina et al., 2007), the current version of ABCN2, and CN2 are described in a technical report found at `https://ailab.si/abml`.

4.2. ABML refinement loop

As it is unlikely that an expert will have time to provide arguments for all learning examples, selecting a subset of examples is important. Therefore, we have developed the *ABML refinement loop*, which picks out such *critical examples* that are likely to help ABML the most. The procedure consists of the following steps:

1. Learn a model with ABML using given data.
2. Find K critical examples and present them to the expert. If no critical examples were found, stop the procedure.
3. Expert provides arguments for a critical example.
4. Add arguments to the selected example in the data.
5. Return to step 1.

4.2.1. Selecting critical examples

Critical examples are learning examples, where arguments lead to a more accurate model. In our previous experiments with ABML (see for example (Groznik et al., 2013)), we simply selected examples with the highest probabilistic error and used them as critical examples.

This approach is problematic, since it will often select outliers. As explaining outliers with arguments will probably not have much impact on the rest of the data, because the data does not contain similar examples, we instead propose to select *prototypical misclassified examples* as critical examples. A prototypical misclassified example is prototypical in the sense of indicating a type of error.

We will now describe our approach for selecting K prototypical misclassified examples. Let $pe(\mathbf{x})$ be probabilistic error of the current model for example $\mathbf{x}$ measured with cross-validation. A prototypical misclassified example should a) have high $pe(x)$ and b) be similar to other examples with high $pe(x)$. Let $sim(\mathbf{x_i}, \mathbf{x_j})$ be a function of similarity between $\mathbf{x_i}$ and $\mathbf{x_j}$. Then, to select K prototypical examples $\mathbf{c_1}, \mathbf{c_2}, \ldots, \mathbf{c_K}$, we need to maximize:

$$\underset{\mathbf{c_1},\mathbf{c_2},\ldots,\mathbf{c_K}}{\operatorname{argmax}} \sum_{i=1}^{K} \sum_{x \in S_i} sim(\mathbf{c_i}, \mathbf{x}) \times pe(\mathbf{x}). \tag{1}$$

Term S_i represents the cluster of learning examples whose closest critical example is the critical example c_i.

If the similarity function is bounded within interval $[0, 1]$ it can be replaced by the distance function, $dist(\mathbf{x_i}, \mathbf{x_j}) = 1 - sim(\mathbf{x_i}, \mathbf{x_j})$, resulting in a criterion that needs to be minimized instead of maximized:

$$\underset{\mathbf{c_1},\mathbf{c_2},\ldots,\mathbf{c_K}}{\operatorname{argmin}} \sum_{i=1}^{K} \sum_{x \in S_i} dist(\mathbf{c_i}, \mathbf{x}) \times pe(\mathbf{x}). \tag{2}$$

This formula is exactly the minimization criterion of the weighted K-center clustering algorithm ($pe(\mathbf{x})$ are weights of instances). We implemented the Lloyd (Lloyd, 1982) clustering algorithm and took the centers found by the algorithm as the critical examples. In this application, we used the standardized Euclidean distance on the binary features (patterns) used in learning.

4.2.2. Argumenting a single critical example

After the selection of critical examples, the expert needs to provide additional domain knowledge for solving a single critical example. The following five steps describe the process that aims at getting as much relevant information as possible.

Step 1: Presenting critical examples to the expert

In this step, critical examples with explanations from the learned model are presented to the domain expert. As critical examples are misclassified by the current model, the current explanation is likely to be wrong. Then, the domain expert is asked the following question: "Why is this example in the class as given?"

Step 2: Adding arguments to the critical example

The expert answers the previous question using natural language without considering domain description of the learning data set. The knowledge engineer then needs to rephrase provided arguments using domain description language (attributes).

However, this is not always possible. In the original idea of ABML (Možina et al., 2007), the experts were supposed to use only existing attributes in explanations of examples. In the following applications of ABML (see for example (Možina et al., 2010; Groznik et al., 2013)) this tool turned out to be an effective tool for selection of new relevant attributes, because experts often related to properties of instances that were not part of the original attribute space. A knowledge engineer then needs to implement the new attribute, or change the definition of an old one.

In this study, the expert also suggested many new attributes. Additionally he also suggested some canonicalizations of examples (programs). For instance, when a program containing the "`not`" operator became critical, the expert suggested to replace this operator with "`\+`", because these two operators are synonyms. After implementing the suggested change, the dimensionality of the problem decreased.

When an expert suggests to change the domain description, implement it and return to Step 1, since the critical example might not be critical anymore. Otherwise, continue to the next step.

Step 3: Pruning arguments

In argumentation, to make an argument stronger and less susceptible to counter-arguments, humans often provide more reasons than are actually needed. In ABML, too many reasons will result in poor generalization.

The method evaluates all reasons in provided arguments whether they are necessary or not. A reason is likely unnecessary if after its removal the argument does not cover any new examples from the opposite class. The expert then needs to decide for each of these reasons, whether to keep them in the argument or not. After removing a single reason from the argument, the pruning procedure is repeated.

Step 4: Discovering counter examples

After arguments are added to the critical example, ABML relearns the complete model using the modified data set. Arguments often apply to many other examples, and not just to the critical example, therefore these arguments will be mentioned in explanations of other examples. When these examples come from the same class as the critical example, such behavior is not problematic, it is in fact favorable, since more examples are now explained using the expert's terms.

On the other hand, if an argument also affects examples from the opposite class, we should check the validity of this argument with the expert. A *counter example* is an example from the opposite class that is consistent with a positive argument provided by the expert, and the induced model mentions this positive argument in explanation of this counter example.

Step 5: Improving arguments using counter examples

When a counter example was found, the expert needs to revise the initial arguments with respect to the counter example. The expert is now asked "Why is critical example in one class and why counter example in the other?" The expert may now revise the original argument and explain the difference between these two examples. Then return to Step 2.

4.3. Learning rules for tutoring

ABCN2 is a general rule-learning algorithm for learning rules from argumented examples. However, not all rules are suitable for representing errors or approaches in programming. In this section, we describe an extension of the algorithm that allows us to extract relevant and explainable patterns from student programs. The extended algorithm is called RL4T (rule learning for tutoring).

A program is incorrect either because a) it contains some incorrect or *buggy* pattern, which needs to be removed or modified, or b) it is missing a relation (pattern) that should be included before the program can be correct. To discover

buggy patterns, we first learn rules for *incorrect* programs called *n-rules*. While learning n-rules, the following constraints are imposed:

1. The classification accuracy (percentage of correct predictions among covered examples) of each learned rule on learning data should exceed a certain threshold (we selected 90% in our experiments, since we deemed 10% error as acceptable).
2. Each conjunct (attribute-value pair) in rule's condition should be significant with respect to the likelihood-ratio test (Clark & Boswell, 1991) (in our experiments significance threshold was set to $p = 0.05$). To validate a condition of a rule we compared rule's class distribution (of covered examples) with the distribution of the rule without this condition.
3. Each conjunct in rule's condition should specify the presence of a pattern, and not absence of a pattern.

The first constraint will lead to learning accurate rules only, hence the conditions of a n-rule will be likely to correspond to a buggy pattern. The second constraint requires that all patterns in a rule's condition are significant, which is necessary to avoid mentioning spurious patterns in explanations. The last constraint assures that the learned rules mention only presence (and not absence) of patterns. Note that the rules stemming from expert's positive arguments were not subjected to these constraints, since these rules were validated by the expert during the ABML refinement loop, as demonstrated in Section 5.1.1.

Allowing only present patterns in rule conditions is important, because rules are meant to be used for explanation. Consider the following example. Let letters $A, B, C, \ldots$ denote different AST patterns of submitted programs for some problem. Let there be two possible solutions of this problem: by implementing patterns A and B, or by implementing C and D. Let E, F, G be three common buggy patterns. Then, if a rule learner may only specify the presence of patterns, three buggy rules can be learned: "if E then *incorrect*", "if F then *incorrect*", and "if G then *incorrect*".

Explaining these three rules is straightforward: if a student implemented

any of the patterns E, F, or G, then the program would be incorrect. On the other hand, if absent patterns were allowed in rules, the following n-rule could appear: "if $\neg A$ and $\neg C$ then *incorrect*". The rule is perfectly valid in a logical sense, since a program cannot be correct if it is missing A and C. However, the explanation of such a rule is ambiguous, because we do not know whether to suggest the student to implement A or C.

To learn rules describing the second type of error, i.e. spotting where students miss important patterns, we could try the same approach and learn rules for the class of correct programs (p-rules). Having accurate rules for correct programs, the conditional part of these rules would define sufficient combinations of patterns that make a program correct.

It turns out that it is difficult to learn accurate rules for correct programs without specifying absent patterns. Consider again our hypothetical example. The accuracy of a rule "if A and B then *correct*" could be significantly lower than the required 90%, as there might be many programs containing A and B, but also including one or more buggy patterns E, F, or G. To produce an accurate rule, the learner would have to include also the absence of buggy patterns, for example: "if A and B and $\neg E$ and $\neg F$ and $\neg G$ then *correct*".

Learning such long rules can be difficult and it is unlikely that rule learner will be able to cover all buggy patterns in p-rules. A possible way to circumvent that is to first remove programs covered by n-rules. This way all known buggy patterns are removed from the data and will therefore not be included in p-rules. However, removing incorrect patterns sometimes removes the need for including relevant patterns in rules. For example, if all incorrect programs were removed, then the single p-rule "if *true* then *correct*" would suffice, which, however, cannot be used to generate explanations.

We achieved the best results by learning p-rules from full data set using the same constraints as above. However, the requirement to achieve 90% classification accuracy was not evaluated on the full learning set, but only on a subset of examples that were not covered by n-rules. Namely, we first learned a set of n-rules, then removed from training set all examples covered by these rules, and

used the remaining examples in the validation of the 90% classification accuracy constraint. Therefore, p-rules can in fact cover many incorrect programs, as long as these programs are also covered by n-rules.

Even though our main interest is discovery of patterns, we can still use induced rules to classify new programs in order to evaluate the quality of rules. The classification procedure has three steps: 1) if an n-rule covers the program, classify it as incorrect; 2) else if a p-rule covers the program, classify it as correct; 3) otherwise, if no rule covers the program, classify it as incorrect.

5. Experiments and Evaluation

In this section we report and discuss the results of learning on a selection of 42 exercises used in our Prolog course. We randomly divided each data set into learning (70%) and testing (30%) set, where all programs submitted by the same student were in the same set. Due to this restriction, the percentage of learning examples was only approximately 70%. Note that all procedures related to the ABML refinement loop, such as using cross-validation for detecting critical examples, can use only learning data.

Initially, the expert (a Prolog teacher) was asked to define a set of relevant patterns for our classification task. He suggested to use patterns that correspond to two occurrences of the same variable, because such patterns characterize the most common constraints in Prolog programs. Both examples illustrated in Figure 1 in Section 3 have this structure.

We learned rules for all 42 problems in the same order as they are presented to students during the lab exercises. We ran the ABML refinement loop only if the initial classification accuracy of learned rules was less than 90%. For example, in the case of the first problem (`sister`), our method scored 98%, hence we skipped the ABML loop. The second problem was the `aunt` problem, where our method initially achieved only 85.2%, and was, therefore, the first problem where argument-based approach was used to elicit knowledge from the domain expert.

In the following section, we give a detailed account of the learning process for the `aunt` problem. At the end of the section, we describe some of the induced rules and explain how we intend to use them to improve learning experience of students. Afterwards, we present results of learning on all 42 problems.

5.1. The `aunt` exercise

During the Prolog course, 336 students tried to solve the `aunt` exercise, submitting 662 programs in total. Typically, students keep submitting programs until they submit a correct program, so there is usually one correct program for each student. Of those 662 programs we used 459 for learning (249 correct, 210 incorrect) and 203 for testing (108 correct, 95 incorrect). The most common correct submission was:

```
aunt(A, B) :-
    parent(C, B),
    sister(A, C).
```

The solution states that `A` is an aunt of `B` if `A` is a sister of a parent of `B`.

There were 57 initial attributes for the `aunt` problem. In the text, we will refer to an attribute with a letter `a` and a number resembling its rank according to the frequency in the learning data: `a0` is the most common attribute, `a1` next, etc. Using these attributes resulted in 85.2% classification accuracy. The model contained 9 n-rules and 3 p-rules. As accuracy was below the specified threshold (90%), we engaged a dialogue between ABML and the domain expert. In the remainder of this section, we present the most interesting interactions between the expert and the algorithm in this problem, show some additional interesting examples from other problems and, at the end, interpret the final results of the `aunt` problem.

5.1.1. A dialogue between ABML and the expert

The first critical example identified by our method was the following correct program:

```
aunt(A, B) :-
    female(A),
    parent(C, B),
    parent(D, A),
    parent(D, C),
    A \== C.
```

The student in this program defined aunt as a female (`female(A)`) whose parent (`parent(D,A)`) is also the grandparent of `B` (`parent(C,B)`,`parent(D,C)`). The student in this case did not think of using the `sister` relation which would make the solution simpler. Our rule learning algorithm, however, failed to learn a rule classifying this program as a correct one.

Following the steps from the Section 4.2, this example was then presented to the expert (Step 3). However, the presentation did not include an explanation, because none of the rules (either n-rules or p-rules) in the initial model covered this example. Then, the expert was asked: "Why is this program correct?" The initial expert argument for this program was: "the program is correct because aunt is a female and her parent (parent of `A`) is the same person as the parent of parent of `B`." The knowledge engineer then used four AST patterns to describe the reasons of this argument (Step 4):

1. Variable `A` from the head is also mentioned in the `female` compound (attribute `a2`):

   ```
   (clause
   (head (compound (functor 'aunt') (args variable)))
   (*and (compound (functor 'female') (args variable))))
   ```

2. The second variable from the head has a parent (attribute `a0`):

   ```
   (clause
   (head (compound (functor 'aunt') (args (args variable))))
   (*and (compound (functor 'parent') (args (args variable)))))
   ```

3. The first variable from the head also has a parent (attribute `a4`):

   ```
   (clause
   (head (compound (functor 'aunt') (args variable)))
   (*and (compound (functor 'parent') (args (args variable)))))
   ```

4. A pattern describing the grandparent relation – some parent has a parent (attribute `a10`):

```
(clause
(*and (compound (functor 'parent') (args variable))
(*and (compound (functor 'parent') (args (args variable)))))
```

The full argument attached to this learning example was the conjunction of the four reasons above. In the following step of the process (Step 3), we found out that the fourth reason (attribute a10) seems to be unnecessary, since programs having the first three reasons, at least in our data set, always contained the fourth reason. After consulting it with the expert, we decided to remove the fourth pattern from the argument.

Using the argument, ABML was able to induce the following new rule:

```
IF a2==T AND a0==T AND a4==T AND a8==T THEN correct [13, 21]
```

The learned rule covers 21 correct programs and 13 incorrect. A careful reader might notice that this rule does not fulfill the first constraint from the RL4T algorithm, as the accuracy is below 90%. Within the definition of the RL4T algorithm, we mentioned that rules AB-covering critical examples do not need to meet these constraints, because ABML could otherwise not be able to find a rule covering the critical example.

The first three conditions embody the above argument by stating that the attributes a2, a0, and a4 must be true (present in the program). The fourth condition (attribute a8) was added by the rule learner. This attribute represents a pattern connecting the first variable from the head and the first variable from the \== goal. In the critical example, this pattern connects the two occurrences of variable A:

```
aunt(A, B) :-
    ...
    A \== C.
```

However, there were many counter examples to the rule presented above. A sample counter example was:

```
aunt(A, B) :-
    parent(C, B),
    parent(D, C),
    parent(D, A),
    female(A),
    A \== B.
```

In this case, a student compared `A` to `B` (`A \== B`) instead of `A` to a parent of `B`. After adding pattern `a14` specifying that a variable from the `\==` goal should also be a parent, the final rule covered 15 correct programs and 8 incorrect:

```
IF a2==T AND a0==T AND a4==T AND a14=T and a8==T THEN correct [8, 15]
```

During the examination of the remaining 8 incorrect examples, we found that 5 of them are covered by n-rules and are therefore not problematic. The remaining three, however, exhibited uncommon mistakes hard to characterize. Therefore, we decided to accept the current rule as good enough and moved on. We will now describe how the remaining critical examples were solved, however with much less detail. The second critical example was a correct program that appears almost the same as the first one:

```
aunt(A, B) :-
    female(A),
    parent(C, B),
    parent(D, A),
    parent(D, C),
    A \= C.
```

In this case, the `\=` inequality operator was used instead of `\==`. The difference is that the former operator checks that the two variables cannot be unified (*i.e.* they cannot be instantiated to the same value), whereas the latter only validates whether left and right sides are not literally equal.

This is the first case where our expert was thinking about canonicalization. The question was whether it makes sense to replace all `\=` operators with `\==`. He decided against it, because in general the distinction between operators is important, although it might work in the case of the `aunt` problem. Instead, we added an almost identical argument as in the case of the first critical example.

The alternative would be to use data-driven canonicalization (Nguyen et al., 2014), where canonicalization is inferred automatically for each problem.

In the third critical example, the student used the *or*(;) connective (disjunctions). It states that aunt is a sister of either father or mother:

```
aunt(A, B) :-
    sister(A, C),
    (father(C, B) ; mother(C, B)),
    female(A).
```

In this case, the expert explained that this is a correct program, because student is referencing mother or father, which is the same as parent. However, as it was difficult to capture the semantics of the *or* clause with an AST pattern, we split all Prolog programs containing the *or* operator into two equivalent clauses:

```
aunt(A, B) :-
    sister(A, C),
    father(C, B),
    female(A).
```

```
aunt(A, B) :-
    sister(A, C),
    mother(C, B)),
    female(A).
```

We implemented the suggested change and moved to the next iteration. It turned out that canonicalization did not solve our previous critical example, since the next critical example was the same. Yet, this does not mean that splitting clauses having *or* operator is generally bad, in fact, it was useful in many other cases. The true problem of the current critical example are relations `father` and `mother`, which are too rare to be present among attributes, as a pattern must occur within five or more programs to become an attribute. We could therefore not explain this example and decided to move to the next critical example.

The fifth critical example also contained the *or* operator, but with `brother` and `sister` goals. Its canonicalized version is:

```
aunt(A, B):-
    female(A),
    parent(C, B),
    brother(C, A).
```

```
aunt(A, B):-
    female(A),
    parent(C, B),
    sister(C, A).
```

The clauses state that aunt `A` is a female that has a brother or a sister who is the parent of `B`. The initial argument why this programs is correct was: "the program is correct because the body states that `A` is a female and she has a sister or a brother". However, adding this argument to the critical example resulted in the following counter-example:

```
aunt(A, B) :-
    female(A),
    parent(C, B),
    brother(C, A) ; sister(C, A).
```

The problem can be seen better with the canonicalized program:

```
aunt(A, B) :-                  aunt(A, B) :-
    female(A),                      sister(C, A).
    parent(C, B),
    brother(C, A).
```

The program lacks parentheses around the part where *or* operator links `brother` and `sister`. After extending the argument with another reason that connects variable `C` from the `parent` goal to variable `C` in the `sister` goal, the algorithm did not find any other counter-examples.

The sixth critical example was almost identical to the first critical example:

```
aunt(A, B) :-
    female(A),
    parent(C, A),
    parent(C, D),
    parent(D, B).
    C \== A.
```

This example is different from the first one, because the variables in goal `C \== A` are in reversed order. Since order is important in pattern representation, the same rule cannot cover both examples. We decided to change the domain description: the patterns containing equality or inequality binary operators should put all arguments on the left side regardless of where they are in the original program. After the change, this example was correctly classified.

There were five more critical examples. However, as arguments in those cases were similar to the arguments shown above, we will not describe the remaining five iterations.

5.1.2. Some interesting examples from other exercises

Most arguments for the `aunt` exercise lead to adding new constraints for rule learning. However, as later demonstrated, we benefited more from introduction of new transformations and inclusion of new attributes, because these additions were useful also for other exercises, and not just for the specific problem. This section demonstrates two such cases.

In the `memb` exercise, the students have to implement a program that checks whether an element is a member of a list or not: `memb(X,L)` is true if `X` is member of list `L`. The first critical example was:

```
memb(A, B) :-
    B = [A|C].

memb(A, B) :-
    B = [D|C],
    memb(A, C).
```

This program was critical, because students usually do not use auxiliary variables, such as `B` in this case, but simply use the expanded list form in the head. Accordingly, the first clause above is equivalently stated as:

```
memb(A, [A|C]).
```

Although these two clauses are semantically equivalent, they are not syntactically equivalent, and therefore result in different patters. To solve this problem, we implemented canonicalization, where such auxiliary variables are removed and replaced by corresponding values.

In the `conc` exercise, students have to implement a program for concatenation of two lists `conc(L1, L2, L3)`, where `L1` and `L2` are concatenated into `L3`. During the execution of ABML refinement loop we identified another interesting problem. The first critical example was, as a surprise to us, the most common correct solution:

```
conc([], A, A).
conc([B|C], D, [B|E]) :-
    conc(C, D, E).
```

Our rule learner was unable to distinguish between the most common correct implementation and incorrect, because the patterns did not include empty list

literals. Until this moment, the patterns were only relating two occurrences of the same variable. As the base case in this exercise must contain the empty list, there was no way for the algorithm to describe the base case. We extended our patterns set with patterns that relate the empty list with another variable from the same clause.

5.1.3. Description of the final rules for the `aunt` *exercise*

The final model contained 10 n-rules and 7 p-rules. The final classification accuracy (after ABML iterative loop) increased from initial 85.2% to 91.1%. The number of AST patterns in the final domain specification was 125.

The n-rule with the highest accuracy covers 17 incorrect programs and no correct programs. The condition part of this rule contains only one AST pattern, which connects the second argument from the head with the second variable from the `sister` goal. Programs with such a pattern state that one's sister is also one's aunt, which is clearly wrong. An example of such a program is, where the buggy pattern is underlined:

```
aunt(A, B) :-
    parent(C, B),
    sister(A, B).
```

The n-rules provide a mechanism to automatically extract typical student errors. By examining all n-rules, a teacher learns about most common detours and wrong attempts made by students, and can perhaps prevent such mistakes by including extracted information in future lectures. An alternative use of n-rules would be to automatically annotate common programming bugs. For example, an intelligent tutoring system could automatically highlight the above pattern in a program that the student is currently debugging and help him or her to spot the error faster.

Let us examine the second most common n-rule. The AST pattern from the condition part of this rule describes a person that is a mother and at the same time has a sister. While a sister of the mother is indeed also an aunt, such representation does not consider all aunts, because sisters of the father are also aunts. An example of such a program:

```
aunt(A, B) :-
    mother(C,B),
    sister(A,C).
```

The most accurate p-rule for the `aunt` problem contains two AST patterns in the condition part and covers 65 correct programs and 1 incorrect. The first AST pattern connects the second argument from the head with the second argument from the `parent` goal, namely `aunt(_, X) & parent(_, X)`, which means that in order to define aunt of `X`, the parent of `X` is relevant. The other AST pattern describes a situation where the second argument in `sister` contains the same variable as the first in `parent`, meaning that the parent must have a sister. An aunt is hence a sister of a parent:

```
aunt(A, B) :-
    parent(C, B),
    sister(A, C).
```

The interesting thing about p-rules is that the condition may not fully define a correct program. There are many possible incorrect definitions of the `aunt` relation that are consistent with the above p-rule. For example, the rule does not explicitly specify that aunt is also a sister to someone, however almost all students who implemented the above two patterns got this part right. The patterns in a p-rule therefore do not mention those parts of the program which students almost always include, but those that distinguish correct programs from incorrect.

Since different p-rules cover different subsets of examples, we say that each p-rule describes a different approach to solving a particular exercise. However, some rules are sometimes very similar and one could argue that they hardly represent different approaches. For example, the second most accurate p-rule is similar to the first rule, but with goals `parent` and `sister` in different order. A human would regard this two approaches as the same, however to a computer, given the definition of our patterns, where order matters, these approaches are different. If the programs were canonicalized by having the goals sorted, only one rule would be enough. On the other hand, another p-rule states that aunt is

a female that has a parent who is also the grandparent of her niece or nephew. This is indeed an alternative approach defining the `aunt` relation.

5.2. Evaluation on 42 Prolog exercises

Table 1 contains results on 42 exercises, arranged in the same order as they were presented to the students during the course. The values in the table are classification accuracies (CA) of models learned on learning set and tested on testing set.

In the complete process of learning all 42 problems, we conducted 13 ABML loops; in the remaining 29 domains the accuracy was already over 90%. We cumulatively added 34 arguments to 34 critical examples. In the process, negative arguments were not needed and one positive argument per example was enough. The arguments implied four major changes to the initial set of attributes and applied four types of canonicalization. The changes to attributes were:

1. In the case of binary operators "=", "==", "\=", and "\==", the side of a particular operand (left/right) is irrelevant. Therefore, the AST patterns containing this operation will always have one argument only.
2. Singleton variables occur only once in the program and therefore can not be represented by patterns, where only two occurrences are considered. We extended our AST pattern set to include all singleton patterns.
3. Sometimes variables are not enough to describe relations, but constants are also needed (*e.g.* empty list literal "[]" or integer constants). We added all constant singletons and pairs of a literal and a variable to the pattern set.
4. The cut operator ("!") prevents Prolog from backtracking and therefore enables some simplifications in the code, such as skipping goals. All patterns extracted from clauses after a cut are additionally flagged with the word "cut".

We implemented the following canonicalization:

Problem	basic	norm	atts	both	args
sister	0.98	0.98	0.98	0.99	0.99
aunt*	0.85	0.85	0.86	0.88	0.91
cousin*	0.87	0.87	0.87	0.88	0.88
ancestor	0.94	0.95	0.96	0.97	0.97
descendant	0.98	0.99	0.98	0.99	0.99
memb*	0.76	0.92	0.88	0.92	0.92
conc*	0.92	0.97	0.93	0.97	0.97
del	0.91	0.98	0.96	0.99	0.99
insert	0.85	0.96	0.88	0.97	0.97
permute	0.78	0.78	0.97	0.96	0.96
min*	0.81	0.77	0.82	0.89	0.89
max*	0.80	0.76	0.87	0.86	0.89
dup	0.75	0.75	0.95	0.96	0.96
rev	0.71	0.71	0.94	0.95	0.95
palindrome*	0.79	0.81	0.79	0.80	0.88
shiftleft	0.95	0.97	0.94	0.98	0.98
shiftright*	0.86	0.87	0.85	0.84	0.89
divide	0.73	0.73	0.97	0.98	0.98
evenlen*	0.62	0.64	0.82	0.85	0.89
sublist	0.90	0.90	0.91	0.92	0.92
sum	0.53	0.51	0.97	0.98	0.98
len	0.92	0.92	0.93	0.93	0.93
count	0.87	0.92	0.89	0.94	0.94
is-sorted	0.78	0.78	0.90	0.94	0.94
sins	0.86	0.88	0.90	0.92	0.92
isort	0.72	0.72	0.96	0.97	0.97
pivoting	0.70	0.83	0.88	0.93	0.93
quick-sort	0.84	0.90	0.88	0.93	0.93
union	0.78	0.86	0.79	0.90	0.90
intersect	0.69	0.68	0.68	0.92	0.92
diff*	0.54	0.61	0.64	0.86	0.86
is-superset	0.56	0.50	0.90	0.90	0.90
subset	0.86	0.94	0.86	0.94	0.94
powerset	0.96	0.96	0.95	0.95	0.95
memberBT*	0.86	0.88	0.85	0.93	0.91
mirrorBT*	0.72	0.72	0.72	0.72	0.72
deleteBT	0.86	0.81	0.87	0.93	0.93
numberBT	0.64	0.64	0.95	0.95	0.95
depthBT*	0.73	0.76	0.74	0.79	0.89
tolistBT	0.86	0.72	0.95	0.96	0.96
memberT	0.89	0.92	0.90	0.92	0.92
getdigits	0.74	0.79	0.74	0.94	0.94
Improvement		0.02	0.08	0.12	0.13

Table 1: Classification accuracies on 42 Prolog exercises. Asterisks denote exercises where we run the ABML loop. Five columns show results of RL4T (rule learning for tutoring) using different representations of data: basic (original attributes), norm (original attributes with canonicalization), atts (new attributes without canonicalization), both (new attributes and canonicalization), and args (both with arguments).

1. Clauses with the semicolon operator (";"), which implements the *or* relation between goals, are replaced with several clauses without the or operator.
2. All binary unification operators ("=") that have a variable on one side, such as `A = B` or `C = []`, are removed from the clause. All other occurrences of this variable are then replaced by the value from the opposite side. In our example, `A` gets replaced by `B` and `C` gets replaced by `[]`, respectively.
3. Predicate names with the same functionality are unified. For example, `member(X, Y)` and `memb(X, Y)` both evaluate whether `X` is a member of `Y`.
4. Negation in Prolog can be expressed either with the "\+" operator or with the "`not`" operator. We replaced all occurrences of the latter with the former.

The five numerical columns in Table 1 contain classification accuracies of argument-based rule learner for tutoring (RL4T) with different preprocessing of data. The first column shows the initial accuracies, where only the basic attributes were used and no canonicalization was applied. The second column contains accuracies of RL4T on canonicalized data with the initial attributes. The bottom value (the row "Improvement") is the average difference between a particular column and the original result in the first column. For example, the canonicalization improved classification accuracy, on average, by 2%. In the third and the fourth column are the results on the data with enhanced attributes without canonicalization (atts) and with canonicalization (both), respectively. The new attributes improve the accuracy by 8%, but the new attributes and canonicalization combined bears even higher accuracy gain, 12% on average. The fifth column presents the results from learning using data and arguments, which additionally increased the accuracy for 1%.

It seems that the main benefit of arguments are the new attributes and required canonicalization, since the arguments improved accuracy only by 1%. It

is, however, important to note that arguments were used only in 13 out of 42 domains. Furthermore, arguments in some cases will not improve accuracy, even though if these arguments have solved a critical example. If a critical example is an outlier, solving it will not increase accuracy, because the likelihood of finding a similar case among testing examples is very slim. On the other hand, arguments are useful in domains that require several conditions in rules, which often poses a problem for myopic rule learners employing greedy algorithms to learn rules. There are six exercises in Table 1 where arguments helped considerably: in the aunt problem arguments improved accuracy by 3%, in max 3%, palindrome 8%, shiftright 5%, evenlen 4%, and depthBT 10%. Often another benefit from arguments is improved explanation.

These results confirm that it is possible to learn accurate rules, which represent abstract descriptions of common key patterns that characterize students' correct approaches and mistakes. Individually, each learned rule is quite accurate, since in Section 4.3 we specified that each learned rule must have at least 90% classification accuracy. Moreover, these rules also achieve high accuracies when they are used together as a general classifier, which indicates that they cover most of the submitted programs. Therefore, using these rules, we can provide feedback for most submitted programs.

In 34 of 42 domains the final classification accuracy of RL4T on testing data exceeded 90%. Of the remaining eight, in seven cases the accuracy was still high and above 85%. The domains where we failed to get over 90% accuracy with ABML were: cousin (88 %), min (89%), max (89%), palindrome (88%), shiftright (89%), evenlen (89%), mirrorBT (72%), and depthBT (89%). One of the reasons why RL4T performed worse in these domains is a large number of specific approaches to solving a problem. When there are several very specific solutions, the AST patterns describing these solutions might be too rare to be represented as attributes. An example of such a solution was shown in the previous section, where `father` relation was used to implement the `aunt` relation.

We got the worst results in the `mirrorBT` exercise, where students have to

flip a binary tree represented as `b(LeftSubtree, Root, RightSubtree)` over the vertical axis through the root node. A sample solution of this exercise is:

```
mirrorBT(nil, nil). % Empty tree
mirrorBT(b(A, B, C), b(D, B, E)) :-
    mirrorBT(A, E), mirrorBT(C, D).
```

The problem is that AST patterns connect only two occurrences of a single variable, which is not enough to sufficiently describe correct programs in the `mirrorBT` case. Instead, the experiments show that if two occurrences of two variables were represented as patterns, the accuracy would increase over 90%. However, since such patterns are more complicated, we decided not to include them, as the general understandability of rules in other domains would decline.

Table 2 contains classification accuracies of some other general machine learning algorithms. RL (rule learning) is the rule learning algorithm that serves as the basic learner for RL4T, but without additional constraints defined in Section 4.2). Rules learned by RL are combined into a linear classifier with a weighted sum, as suggested by Friedman & Popescu (2008). Other methods are random forests (RF), logistic regression (LR) and majority classifier (MAJ), as implemented in the Orange data mining library (Demšar et al., 2013). These methods were tested with two versions of the same data set; in "orig" we used the original set of attributes, which were defined before we ran the ABML refinement loop, whereas "both" represents the final data, namely canonicalized data with all the attributes. As an alternative way to extract patterns we applied the gSpan algorithm (Yan & Han, 2002), a general frequent subgraph mining algorithm. With gSpan we generated all subgraphs that occurred in at least 5 submissions, and used 10000 most common subgraphs as the attributes for learning in random forests. The last column in Table 2 contains classification accuracies of this approach.

According to the Friedman test, the differences between ranks are significant ($p < 0.001$). Figure 2 contains critical differences between ranks of different methods (Demsar, 2006). Models learned on the final data ("both") are significantly more accurate than models learned on the data with the original

Problem	MAJ	RL4T	RL		LR		RF		
	full	both+args	basic	both	basic	both	basic	both	gSpan
sister	0.72	0.99	0.98	0.99	0.98	0.99	0.98	0.98	0.98
aunt	0.53	0.91	0.96	0.95	0.96	0.95	0.96	0.96	0.94
cousin	0.67	0.88	0.89	0.90	0.88	0.90	0.90	0.89	0.88
ancestor	0.64	0.97	0.95	0.99	0.95	0.98	0.98	0.99	0.98
descendant	0.55	0.99	0.98	0.98	0.98	0.98	0.99	0.99	0.98
memb	0.53	0.92	0.90	0.98	0.90	0.97	0.91	0.98	0.92
conc	0.77	0.97	0.95	0.97	0.95	0.96	0.97	0.99	0.93
del	0.64	0.99	0.97	0.98	0.97	0.98	0.96	0.99	0.94
insert	0.51	0.97	0.95	0.97	0.96	0.97	0.97	0.98	0.92
permute	0.75	0.96	0.92	0.96	0.92	0.96	0.94	0.98	0.94
min	0.72	0.89	0.89	0.94	0.89	0.93	0.87	0.94	0.90
max	0.37	0.89	0.85	0.92	0.85	0.91	0.86	0.95	0.79
dup	0.75	0.96	0.86	0.97	0.86	0.96	0.85	0.97	0.94
rev	0.71	0.95	0.90	0.96	0.89	0.96	0.89	0.95	0.95
palindrome	0.61	0.88	0.76	0.88	0.76	0.88	0.79	0.91	0.89
shiftleft	0.55	0.98	0.97	0.97	0.96	0.96	0.96	0.98	0.93
shiftright	0.49	0.89	0.91	0.94	0.90	0.94	0.90	0.95	0.89
divide	0.73	0.98	0.80	0.99	0.80	0.99	0.81	0.98	0.97
evenlen	0.62	0.89	0.82	0.93	0.82	0.92	0.80	0.95	0.90
sublist	0.71	0.92	0.91	0.95	0.90	0.94	0.96	0.96	0.92
sum	0.51	0.98	0.90	0.99	0.90	0.99	0.89	0.98	0.95
len	0.55	0.93	0.95	0.99	0.95	0.99	0.95	0.99	0.93
count	0.76	0.94	0.91	0.96	0.91	0.96	0.89	0.94	0.88
is-sorted	0.78	0.94	0.78	0.97	0.78	0.98	0.78	0.97	0.94
sins	0.81	0.92	0.90	0.95	0.90	0.95	0.90	0.95	0.93
isort	0.72	0.97	0.87	0.97	0.87	0.97	0.86	1.00	0.97
pivoting	0.69	0.93	0.90	0.97	0.90	0.97	0.90	0.97	0.91
quick-sort	0.63	0.93	0.92	0.97	0.88	0.95	0.91	0.98	0.92
union	0.78	0.90	0.83	0.95	0.83	0.94	0.83	0.94	0.83
intersect	0.67	0.92	0.81	0.96	0.81	0.96	0.76	0.96	0.82
diff	0.54	0.86	0.76	0.89	0.75	0.89	0.79	0.90	0.81
is-superset	0.44	0.90	0.82	0.89	0.82	0.86	0.83	0.90	0.92
subset	0.74	0.94	0.91	0.95	0.91	0.96	0.93	0.96	0.95
powerset	0.54	0.95	0.96	0.97	0.96	0.97	0.96	0.97	0.96
memberBT	0.55	0.91	0.93	0.96	0.93	0.96	0.95	0.97	0.91
mirrorBT	0.72	0.72	0.83	0.96	0.83	0.96	0.83	0.94	0.95
deleteBT	0.78	0.93	0.89	0.94	0.89	0.94	0.84	0.93	0.90
numberBT	0.64	0.95	0.86	0.99	0.86	1.00	0.85	0.99	0.95
depthBT	0.59	0.89	0.77	0.87	0.77	0.87	0.80	0.91	0.86
tolistBT	0.72	0.96	0.93	0.97	0.93	0.97	0.92	0.97	0.95
memberT	0.79	0.92	0.90	0.91	0.90	0.94	0.91	0.91	0.90
getdigits	0.68	0.94	0.80	0.95	0.80	0.95	0.80	0.98	0.90
Avg. rank	8.99	4.52	6.19	2.55	6.63	2.79	5.88	2.05	5.40

Table 2: A comparison of RL4T, rule learning (RL), and general machine learning methods: random forests (RF), logistic regression (LR) and majority classifier (MAJ). RL, LR, and RF were tested on the data with original attributes and on the data with both improvements (new attributes and canonicalization). RF were also tested on data, were patterns were automatically extracted with gSpan, a generic algorithm for frequent subgraph discovery.

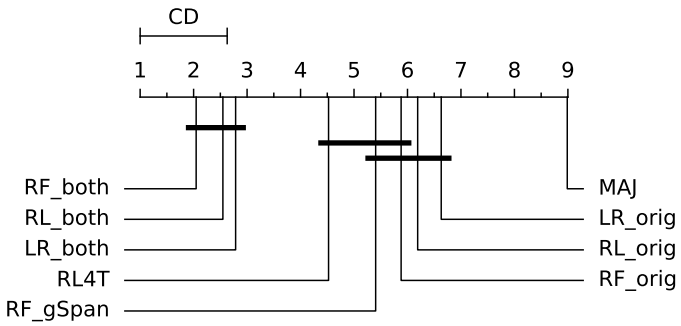

Figure 2: Critical differences of ranks in Table 2. We used Bonferroni-Dunn test at $\alpha = 0.05$.

attributes. This result shows a general usefulness of ABML as a knowledge elicitation tool, since the new attributes and canonicalization also help other methods, and not just the method that was used in the loop.

The accuracy of rules induced for tutoring (RL4T) is significantly lower than the accuracy of the RL model, which is the rule learner used within RL4T. The reasons are the constraints imposed on RL4T in Section 4.3. As explained, these constraints are necessary to increase the interpretability of rules. RL, on the other hand, does not need to conform to these requirements. As a consequence, for example, the problematic `mirrorBT` exercise becomes solvable when rule conditions can specify the absence of patterns, as the accuracy of RL is 96% instead of 72% achieved by RL4T.

Logistic regression (LR) performs the worst, although not significantly, among standard machine learning algorithms RL, RF and LR. A linear combination of patterns therefore seems to be insufficient, and more complex combinations, such as conjunctions of patterns, perform better. This result is expected, because each pattern represents a basic constraint in a Prolog program. To construct a correct program, one needs to combine several of these patterns.

Random forest using the final AST patterns (RF_both) performs significantly better than the random forest using patterns extracted with gSpan (RF_gSpan). The gSpan patterns, on the other hand, seem to be better than the patterns

that were originally suggested by the teacher. However, we must emphasize here that we implemented the most basic and model-independent procedure for pattern selection and applying a more sophisticated procedure, such as iterative extraction of patterns (Bringmann et al., 2011), would likely result in more accurate models. Nevertheless, even if the accuracy of automatically extracted patterns was better, it would still be difficult to use such model in explanations, as automatically extracted patterns tend to be less comprehensible to humans than patterns that were defined by them.

6. Conclusions

We have described a process for learning rules that characterize typical approaches and errors in students' programming solutions. We described an argument-based rule learning algorithm that was tailored for this task, and an extended version of the ABML loop for acquiring arguments from experts. The most important extensions are the algorithm for selecting critical examples and the algorithm for selecting counter examples.

We evaluated the approach on our educational data of Prolog programs. Each program was encoded with a set of patterns extracted from the abstract syntax tree. In the first part of evaluation, we demonstrated the effectiveness of the ABML loop, which enabled the expert provide exactly the information that was missing from the data. We have shown that arguments sometimes mention existing attributes, but sometimes we had to implement several new attributes (patterns) and canonicalizations of AST trees as a result. In the second part of evaluation, we presented a 13% increase of classification accuracy of learned model averaged over all problems.

In addition to the algorithms, the case study illustrates three important points in the context of programs data-mining. First, representing programs with AST patterns seems a viable strategy, because we were able to achieve high classification accuracies (over 90%) for a large majority of problems. Second, even though the space of possible AST patterns is large, we were able to select

a relatively small subset of relevant patterns with the ABML loop. Third, since AST patterns are interpretable, the learned rules are also interpretable. We can therefore use them to characterize typical approaches and errors in programming by humans.

Probably the main weakness of ABML is that it requires a lot of time to produce the final model. However, the source of this problem is not the computational complexity of the algorithms, as one would initially assume, but the time that experts need to express their arguments together with the time that knowledge engineers need to implement their arguments. In the presented case study, the time spent by algorithms was negligible (a few seconds to find a critical example) when compared to the time spent by the experts. The refinement loop is designed to reduce the effort of experts as much as possible by "smart" selection of critical examples. In this application, 34 critical examples were examined. In most of the cases, the critical examples were processed relatively quickly (less than 1h for one critical example). However, when a more complex operation was required, say implementing a new canonicalization procedure, then the process had to stop until the knowledge engineer implemented the necessary changes. Both quicker alternatives, namely to ask the experts in advance for all necessary patterns or automatic extraction of patterns, failed to provide satisfying result. Though some effort is still required, we therefore regard ABML as an effective knowledge elicitation tool.

As mentioned in the introduction, learned rules can potentially be used in various applications. We have implemented automatic hint generation based on rules in the CodeQ[3] tutor for teaching Prolog programming. Furthermore, the teachers were able to identify and understand several interesting patterns of student programming, which can influence their teaching.

We have gathered similar data for Python in our CodeQ tutoring application. We intend to apply the same procedure to learn patterns in Python programs. However, since Python's syntax is considerably less constrained than Prolog's

[3] `https://codeq.si`

and Python programs are usually longer, we anticipate this learning problem to be more complicated. We believe that the structure of AST patterns will stay the same, but they will have to be more sophisticated. For example, we might be required to store additional context information related to nesting of control statements. Nevertheless, with the ABML loop we should be able to identify which information is required to learn accurate rules for Python.

In procedural languages, such as Python, we could further extend the set of patterns with dynamic information. For example, we could record execution traces of variables as presented in the Online Python Tutor (Guo, 2013) and then extract patterns from these traces. Another option is to include patterns from linkage graphs Jin et al. (2012). Since both types of patterns are complementary to static AST patterns, we think that this could improve the accuracy and expressiveness of induced rules.

In this paper and in previous publications, ABML and the refinement loop were presented as knowledge elicitation tools for rule-based classification models. How to apply ABML to other machine learning algorithms remains an open question. Another open question is the selection of critical examples, which is probably the most important step of the refinement loop. Would it be possible to evaluate the usefulness of critical examples? Would presenting other critical examples decrease or increase the number of interactions between ABML and an expert?

The problem of finding informative patterns for teaching programming is closely related to software fault localization Wong et al. (2016), where one tries to identify which parts of the programming code more likely contain a bug. Due to the increasing complexity of software programs, this area is recently receiving more attention. The problem of finding a bug can be presented as a machine learning problem, where the goal is to either uncover static or dynamic (execution) patterns that suggest a bug. Since test cases are used as learning instances and usually there are only a few available, human intervention is often required to define patterns or new test cases. Given that they use similar data for learning (programming code), we believe that the data representation and the

techniques presented in this paper could be useful for software fault localization.

Acknowledgements

This work was partly supported by the Slovenian Research Agency (ARRS).

References

Amershi, S., Fogarty, J., Kapoor, A., & Tan, D. (2010). Examining multiple potential models in end-user interactive concept learning. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (pp. 1357–1360). ACM.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (1990). Cognitive modeling and intelligent tutoring. *Artificial Intelligence*, *42*, 7–42.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the list tutor. *Cognitive Science*, *13*, 467–505.

Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to one tutoring. *Educational Researcher*, *13*, 4–16.

Bringmann, B., Nijssen, S., & Zimmermann, A. (2011). Pattern-based classification: A unifying perspective. *CoRR*, *abs/1111.6191*.

Clark, P., & Boswell, R. (1991). Rule induction with CN2: Some recent improvements. In *Machine Learning - Proceeding of the Fifth Europen Conference (EWSL-91)* (pp. 151–163). Berlin.

Demšar, J., Curk, T., Erjavec, A., Črt Gorup, Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., & Zupan, B. (2013). Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, *14*, 2349–2353. URL: `http://jmlr.org/papers/v14/demsar13a.html`.

Demsar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, *7*, 1–30.

Domingos, P. (2007). Toward knowledge-rich data mining. *Journal of Data Mining and Knowledge Discovery*, *15*, 21–28.

Fails, J. A., & Olsen, D. R., Jr. (2003). Interactive machine learning. In *Proceedings of the 8th International Conference on Intelligent User Interfaces* IUI '03 (pp. 39–45).

Folsom-Kovarik, J. T., Schatz, S., & Nicholson, D. (2010). Plan ahead: Pricing ITS learner models. In *Proc. 19th Behavior Representation in Modeling & Simulation Conference* (pp. 47–54).

Friedman, J. H., & Popescu, B. E. (2008). Predictive learning via rule ensembles. *The Annals of Applied Statistics*, *2*, 916–954.

Gerdes, A., Heeren, B., Jeuring, J., & van Binsbergen, L. T. (2017). An adaptable programming tutor for haskell giving automated feedback. *Internation Journal of Artificial Intelligence in Education*, *27*, 65–100.

Glassman, E. L., Scott, J., Singh, R., Guo, P. J., & Miller, R. C. (2015). Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction*, *22*, 1–35.

Grossi, V., Romei, A., & Turini, F. (2017). Survey on using constraints in data mining. *Data Mining and Knowledge Discovery*, *31*, 424–464.

Groznik, V., Guid, M., Sadikov, A., Možina, M., Georgiev, D., Kragelj, V., Ribarič, S., Pirtošek, Z., & Bratko, I. (2013). Elicitation of neurological knowledge with argument-based machine learning. *Artificial intelligence in medicine*, *57*, 133–144.

Guid, M., Možina, M., Groznik, V., Sadikov, A., Georgijev, D., Pirtošek, Z., & Bratko, I. (2012). Abml knowledge refinement loop: A case study. In

Proceedings of the 2012 IEEE 20th International Symposium (ISMIS 2012) (pp. 41–50).

Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for cs education. In *In Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education*.

Holland, J., Mitrovic, A., & Martin, B. (2009). J-latte: a constraint-based tutor for java. In *Proceedings of 17th International Conference on Computers in Education* (pp. 142–146). Hong Kong.

Jiang, C., Coenen, F., & Zito, M. (2013). A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, *28*, 75–105.

Jin, W., Barnes, T., Stamper, J., Eagle, M. J., Johnson, M. W., & Lehmann, L. (2012). Program representation for automatic hint generation for a data-driven novice programming tutor. In *Proc. 11th Int'l Conf. Intelligent Tutoring Systems (ITS 12)* (pp. 304–309).

Ketkar, N. S., Holder, L. B., & Cook, D. J. (2009). Empirical comparison of graph classification algorithms. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining* (pp. 259–266). Nashville, USA.

Keuning, H., Jeuring, J., & Heeren, B. (2016). Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 41–46). ACM.

Koedinger, K. R., Aleven, V., & Hefferman, N. (2003). Toward a rapid development environment for cognitive tutors. In *In Proceedings of the International Conference on Artificial Intelligence in Education* (pp. 455–457). IOS Press.

Koedinger, K. R., Aleven, V., Hefferman, N., McLaren, B., & Hockenberry, M. (2004). Opening the door to non-programers: Authoring intelligent tutor behaviour by demonstration. In *Intelligent Tutoring System* (pp. 162–174).

Koedinger, K. R., & Anderson, J. R. (1997). Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education*, *8*, 30–43.

Kulesza, T., Burnett, M., Wong, W.-K., & Stumpf, S. (2015). Principles of explanatory debugging to personalize interactive machine learning. In *Proceedings of the 20th International Conference on Intelligent User Interfaces* IUI '15 (pp. 126–137).

Langley, P., & Simon, H. A. (1995). Applications of machine learning and rule induction. *Communications of the ACM*, *38*, 54–64.

Lazar, T., Mozina, M., & Bratko, I. (2017). Automatic extraction of AST patterns for debugging student programs. In *Artificial Intelligence in Education - 18th International Conference, AIED 2017, Wuhan, China* (pp. 162–174).

Le, N.-T. (2016). A classification of adaptive feedback in educational systems for programming. *A Classification of Adaptive Feedback of Educational Systems for Programming*, *4*.

Le, N.-T., Loll, F., & Pinkwart, N. (2013). Operationalizing the continuum between well-defined and ill-defined problems for educational technology. *IEEE Transactions on Learning Technologies*, *6*, 258–270.

Le, N.-T., Menzel, W., & Pinkwart, N. (2009). Evaluation of a constraint-based homework assistance system for logic programming. In *Proceedings of 17th International Conference on Computers in Education* (pp. 51–58). Hong Kong.

Levy, R., & Andrew, G. (2006). Tregex and tsurgeon: tools for querying and manipulating tree data structures. In *5th International Conference on Language Resources and Evaluation (LREC 2006)* (p. 2231–2234).

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE Trans. Inf. Theor.*, *28*, 129–137.

Matsuda, N., Cohen, W. W., & Koedinger, K. R. (2015). Teaching the teacher: Tutoring simstudent leads to more effective cognitive tutor authoring. *International Journal of Artificial Intelligence in Education*, *25*, 1–34.

Mitrovic, A. (2012). Fifteen years of constraint-based tutors: what we have achieved and where we are going. *User Modeling and User-Adapted Interaction*, *22*, 39–72.

Možina, M., Guid, M., Krivec, J., & Sadikov, A. (2010). Learning to explain with abml. In *Proceedings of the 5th International Workshop on Explanation-aware Computing (Exact 2010)* (pp. 37–48). Lisbon, Portugal.

Možina, M., Žabkar, J., & Bratko, I. (2007). Argument-based machine learning. *Artificial Intelligence*, *171*, 922–937.

Murray, T. (1999). Authoring intelligent tutoring systems: An analysis of the state of the art. *Journal of Artificial Intelligence in Education*, *10*, 98–129.

Nguyen, A., Piech, C., Huang, J., & Guibas, L. (2014). Codewebs: scalable homework search for massive open online programming courses. In *Proc. 23rd Int'l World Wide Web Conf. (WWW 14)* (pp. 491–502).

Nkambou, R., Fournier-Viger, P., & Nguifo, E. M. (2011). Learning task models in ill-defined domain using a hybrid knowledge discovery framework. *Knowledge-Based Systems*, *24*, 176–185.

Ohlsson, S. (1992). Constraint-based student modeling. *Journal of Artificial Intelligence in Education*, *3*, 429–447.

Piech, C., Sahami, M., Huang, J., & Guibas, L. (2015). Autonomously generating hints by inferring problem solving policies. In *Proc. 2nd ACM Conference on Learning @ Scale (L@S 2015)* (pp. 195–204). ACM.

Rivers, K., & Koedinger, K. R. (2015). Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor. *International Journal of Artificial Intelligence in Education*, (pp. 1–28).

Stumpf, S., Rajaram, V., Li, L., Wong, W.-K., Burnett, M., Dietterich, T., Sullivan, E., & Herlocker, J. (2009). Interacting meaningfully with machine learning systems: Three experiments. *Int. J. Hum.-Comput. Stud.*, *67*, 639–662.

Suarez, M., & Sison, R. (2008). Automatic construction of a bug library for object-oriented novice java programmer errors. In *Intelligent Tutoring Systems* (pp. 184–193). Heidelberg: Springer.

Suraweera, P., Mitrovic, A., & Martin, B. (2010). Widening the knowledge acquisition bottleneck for constraint-based tutors. *International Journal of Artificial Intelligence in Education*, *20*, 137–173.

VanLehn, K. (2011). The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, *46*, 197–221.

Voosen, P. (2017). The ai detectives. *Science*, *357*, 22–27.

Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, *42*, 707–740.

Yan, X., & Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining* ICDM '02 (pp. 721–). Washington, DC, USA: IEEE Computer Society.