Designing an Interactive Teaching Tool with ABML Knowledge Refinement Loop

Matej Zapušek[1], Martin Možina[2], Ivan Bratko[2], Jože Rugelj[1], Matej Guid[2]

[1] Faculty of Education, University of Ljubljana, Slovenia
[2] Faculty of Computer and Information Science, University of Ljubljana, Slovenia

Abstract. Argument-based machine learning (ABML) knowledge refinement loop offers a powerful knowledge elicitation tool, suitable for obtaining expert knowledge in difficult domains. In this paper, we first use it to conceptualize a difficult, even ill-defined concept: distinguishing between "basic" and "advanced" programming style in python programming language, and then to teach this concept in an interactive learning session between a student and the computer. We demonstrate that by automatically selecting relevant examples and counter examples to be explained by the student, the ABML knowledge refinement loop provides a valuable interactive teaching tool.

Keywords: intelligent tutoring, knowledge elicitation, argument-based machine learning, ill-defined concept, programming style, computer programming, python

1 Introduction

Argument-based machine learning (ABML) knowledge refinement loop offers a powerful knowledge elicitation tool, suitable for obtaining expert knowledge in difficult domains [2, 3, 6]. Benefits of ABML for knowledge elicitation include: (1) the expert only needs to explain a single example at the time, (2) it enables the expert to provide most relevant knowledge by showing him problematic examples only, and (3) it helps the expert to detect deficiencies in his or her explanations by providing counter examples [3]. In this paper, we would like to verify whether ABML knowledge refinement loop could also be used by students, as an interactive teaching tool based on machine learning and argumentation.

As our case study we selected a difficult, hard to define concept: programming style in python programming language. This language often enables short and elegant solutions. And although the meaning of this latter word is not well defined, it is quite widely accepted in computer programming what has been nicely put by Richard O'Keefe: "Elegance is not optional." [8]

We were particularly interested in distinguishing between "basic" and "advanced" solutions of exercises that typically occur in introductory programming lessons with python as the language of choice. Consider the following solutions:

```
# Solution 1
def most_different(words):
    most_letters = 0
    for word in words:
        characters = []
        for c in word.lower():
            if not c in characters:
                characters.append(c)
        if len(characters) > most_letters:
            most_letters = len(characters)
            most_diff_word = word
    return most_diff_word
```

```
# Solution 2
def most_different(words):
    return max(words, key=lambda x:len(set(x.lower())))
```

Fig. 1. A "basic" solution (left) and an "advanced" solution (right).

Although both solutions apply to the same exercise, they demonstrate two very different approaches to solve it. In both cases the problem is divided into several subproblems. However, in Solution 1 each subproblem is expressed separately, while Solution 2 effectively utilizes available built-in functions and mechanisms. The first solution (left) is less sophisticated and clearly a preferred option for the beginners, while the second one (right) is arguably more elegant, more advanced, and perhaps even easier to read by an advanced programmer, but may be difficult to understand for beginners.

While this paper is *not* concerned whether the second solution is better than the first one, our domain expert – a teacher of introductory programming course – labeled solutions such as Solution 1 as "basic," and solutions such as Solution 2 as "advanced." Our goal was to design an interactive tool for supporting students to learn this concept, with respect to distinguishing advanced solutions from the basic ones.

The experts in this domain are generally able to recognize good or bad programming style merely by observing solutions, provided that the solutions are correct, sensible, and complex enough to enable a more advanced approach [1]. The expert should therefore be able to distinguish between "basic" and "advanced" programming style (i.e., between simple and more sophisticated solutions) based on solutions only. In our approach, the text of the exercise did not influence the expert's decisions at all. Our teaching tool should therefore not depend on understanding semantics (or deeper meanings) of the exercise text.

Note that the aim of this paper is *not* to debate what is a suitable programming style and whether the recognition of "elegant" or "advanced" programming style is possible by observing the solutions of programming exercises only (without knowing the instructions of the exercise itself). Nor do we claim that our teacher's views about programming style in python programming language are absolutely correct or indisputable. The goal of this paper is merely to demonstrate the use of argument-based machine learning (ABML) knowledge refinement loop for the purpose of designing an interactive teaching tool. In particular, we intend to demonstrate the use of ABML knowledge refinement loop for: (1) knowledge elicitation of a difficult (even ill-defined) concept from the domain expert – a teacher of introductory computer programming, and (2) student-computer interaction that involves student's argumentation of automatically selected examples and counter examples.

A similar idea, however with a different goal, was explored in a system for smart authoring of automated tutors, *SimStudent*, where students can learn by teaching a live machine-learning agent, using a game-like learning environment [5]. Nan *et* al. showed that an extended version of *SimStudent* successfully learns grammar rules for the difficult task of article selection in English [4].

The paper is organized as follows. In Section 2, we briefly explain the experimental design. Section 3 highlights two important goals of knowledge elicitation from the teacher, namely to obtain (1) relevant description language in the form of new attributes, and (2) consistently labeled learning data. In Section 4, we describe in detail the interactive learning session between a student and the computer, using our (argument-based) teaching tool. Also, the results of an experiment with students learning to distinguish between basic and advanced solutions are presented. We then conclude the paper and point out directions for future work.

2 Experimental Design

From a textbook of introductory programming in python, we selected 121 solutions of 62 different exercises. The teacher labeled each solution as "basic," or "advanced." We randomly selected 91 solutions for learning and 30 solutions for testing (the proportion of positive and negative examples was preserved).

In order to design a successful teaching tool, it was first required to "conceptualize" the domain, that is, to elicitate relevant knowledge from the teacher and transform it into both human- and computer-understandable form. Also, the labels of examples had to be corrected, if necessary. The knowledge in form of attribute values and correct labels had to be incorporated into the teaching tool. Finally, the teaching tool had to be tested by the students. At the end of the interactive session, the students were therefore asked to classify all 30 examples in the test set.

The teaching tool was operated by the teacher. It is essentially based on ABML knowledge refinement loop, and has the following main properties:

1. It is capable of building a rule-based model, using attributes and arguments that are currently included into the domain.
2. It finds "critical examples," i.e. examples that the current model cannot classify successfully, and therefore should be explained.
3. It enables the user to explain given examples in various ways:
   - by introducing (predefined) attributes into the domain,
   - by attaching arguments to selected critical examples,
   - by assigning constraints to particular attributes in the arguments (*high*, *low*, *true*, *false*, *higher*/*lower* than a particular value etc.)
4. It selects appropriate "counter examples," if necessary.
5. It measures the progress of the student (in terms of accuracy of the obtained rules on the unseen test data). However, this information was not disclosed to the students during the experiments.

A detailed description of the ABML knowledge refinement loop can be found in [3] and [7].

3 Knowledge Elicitation from the Teacher

The knowledge elicitation process is described in detail in the next section, where the student-computer interaction is presented. It is actually very similar to that interaction, however, there are two very important differences:

- features (attributes) that would describe the domain well are not yet known,
- labels of examples (given by the teacher) are likely to contain inconsistences.

The goal of the knowledge elicitation from the teacher is therefore not only to obtain a (rule-based) model consistent with his knowledge, but – even more importantly – (1) to obtain relevant description language in the form of new attributes, and (2) to obtain consistently labeled learning data.

This goal is achieved with the help or relevant critical examples and counter examples being presented to the teacher during the interaction. As the teacher is asked to explain given examples or to compare the critical examples to the counter examples, he may introduce new attributes into the domain. The inconsistences are usually found quite easily, since inconsistently labeled examples are likely to appear as critical or counter examples.

In the present case study, the knowledge elicitation process consisted of 9 iterations. Table 1 shows the list of all attributes used: at the beginning of the process 5 of them were included into the domain, and 9 new attributes were introduced by the teacher during the process. Only 1 initial attribute remained in the final model.

Table 1. List of attributes.

#	Attribute	Type	Description	Start	Final
1	*cRows*	cont.	number of rows	X	X
2	*cVar*	cont.	number of variables	X	
3	*cFor*	cont.	number of `for` loops	X	
4	*cWhile*	cont.	number of `while` loops	X	
5	*cIf*	cont.	number of conditionals	X	
6	*NeLoop*	T/F	occurrence of nested loop		X
7	*LiCom*	T/F	occurrence of list comprehension		X
8	*cLCbFor*	cont.	number of tokens before the last `for` in list compr.		X
9	*cLCaFor*	cont.	number of tokens after the last `for` in list compr.		X
10	*Zip*	T/F	occurrence of `zip` function		X
11	*cSlice*	cont.	number of list slices		X
12	*Lambda*	T/F	occurrence of `lambda` function		X
13	*cFunc*	cont.	number of built-in functions		X
14	*cMeth*	cont.	number of built-in methods		X

The attributes that occurred in the rules of the final model were included in the interactive teaching tool presented in the next section. The final model contained 9 rules, all of them were found sensible by the expert.

4 A Student-Computer Interactive Learning Session

At the start of the learning session, each student is given the following task: to obtain rules for determining whether a particular solution of (an unknown) programming exercise is an advanced one. The rules must consist of the attributes that remained in the final model obtained by the teacher (see Table 1) only. That is, the goal of the interaction is the student being able to express the target concept using the teacher's expressive language. The instructions were accompanied with a simple example that demonstrates differentiating between a basic and an advanced solution, similar to the one in Fig. 1. In order to facilitate learning, the ABML knowledge refinement loop was used to present the student with relevant examples (and counter examples, if necessary). To accomplish the task in as few iterations as possible, the students are advised to give explanations that:

- contain the most important feature(s) to explain the given example,
- use the smallest possible number of features in a single argument,
- try not to repeat the same arguments.

In the sequel, we demonstrate 4 out of 5 iterations of a typical interaction that actually occurred in one of the learning sessions.

Iteration 1 In the beginning of the interaction, only 5 initial attributes listed in Table 1 were included into the domain, and no arguments were given yet. The solution *A.20-3* (Fig. 2) was the first critical example presented to the student. The student was asked to explain which features speak in favor of this solution being an advanced one. His argument was "the solution is *advanced* because function `zip` is present and the number of rows is low." He also gave an interesting remark that the overall number of different tokens in the solution might have been a more appropriate feature than simply the number of rows.

```
# solution A.20-3 (advanced)
def crossword(word, words):
    return [d for d in words if len(d) == len(word) \\
        and all(c1 == '.' or c1 == c2 for c1, c2 in zip(word, d))]

# solution B.14-1 (basic)
def match(b1, b2):
    b = ""
    for c1, c2 in zip(b1, b2):
        b += c1 if c1 == c2 else "."
    return b
```

Fig. 2. The first "critical example," and the corresponding "counter example."

The student's argument was not sufficiently good: the algorithm selected the (basic) solution *B.14-1* (see Fig. 2) as the counter example. He was asked to compare the counter example *B.14-1* with the critical example *A.20-3* and try to improve the argument. The student noticed an important difference between

the two examples: the advanced solution *A.20-3* contains a list comprehension, whereas the basic solution *B.14-1* does not. He extended the argument to "the solution is *advanced* because `zip` function is present, the number of rows is low, and a list comprehension occurs." There were no more counter examples.

Iteration 2 The (advanced) solution *A.35-1* (see Fig. 3) was then presented to the student. The student now observed a relatively high number of occurring methods (join, split, lower) and chose this as the most important argument. Again he gave an interesting suggestion: namely, that the attribute *cMeth* should have been normalized, taking into account the overall number of tokens in the solution. Another suggestion was to include a new feature: the number of *distinct* methods that occur in the solution.

```
# solution A.35-1 (advanced)
def censorship(text, forbidden):
    return " ".join(word for word in text.split() \\
        if word.lower() not in forbidden)
```

Fig. 3. Solution with "a high number of occurring methods and a list comprehension."

The method now selected a solution from the class "basic" as the counter example. The student quickly noticed several differences between the two solutions, and chose the fact that the advanced one contains a comprehension list to be the most important one among them. The argument was thus extended to "the solution is *advanced* because the number of used methods is high, and a list comprehension occurs.". The algorithm did not find more counter examples.

Iteration 3 was very similar to the second one, thus we skip its description.

Iteration 4 The solution *A.13-3* (see Fig. 4), again an advanced one according to the teacher, was presented to the student. He now selected a new attribute to describe the reasons for the teacher's opinion: the relatively high number of occurring functions.

```
# solution A.13-3 (advanced)
def pairs():
    return [(i, j) for i in range(1, 101) for j in range(i+1, 1001) \\
        if len(str(i)) != len(str(j)) and sum(map(int, str(i))
```

Fig. 4. Another python "one-liner."

After another counter example, the student extended his argument with another unused attribute from the list of available features: a relatively high number of tokens *after* the last `for` statement within the list comprehension. The argument was extended to "the solution is *advanced* because the number of used functions is high, and the number of tokens after the last `for` in the list comprehension is high.". He also suggested that the number of `for` statements within a list comprehension would be another interesting attribute. The extension of the argument worked well: no counter examples were found.

Iteration 5 The student's arguments now resulted in a rule-based model that covered all positive examples for the class "advanced". However, now the algorithm found a problematic example of a different kind: a basic solution that was also covered by one of the obtained rules for the opposite class. The problematic example was the solution *B.34-2*. The key example of the problematic rule was the solution *A.20-2*. The student was now asked to compare these two solutions (see Fig. 5). Namely, what makes the solution *A.20-2* more advanced compared to the solution *B.34-2*.

```
# solution B.34-2 (basic)
def even_vs_odd(s):
    t = sum(e % 2 for e in s) > len(s) / 2
    return [e for e in s if e % 2 == t]

# solution A.20-2 (advanced)
def crossword(word, words):
    return [d for d in words if len(d) == len(word) \\
        and all(c1 == '.' or c1 == c2 for c1, c2 in zip(word, d))]
```

Fig. 5. What makes the solution *A.20-2* more advanced compared to *B.34-2*?

The student chose another yet unused attribute from the list of available features: a relatively high number of tokens *before* the last `for` statement within the list comprehension. After seeing a relevant counter example he extended the argument with the presence of the `zip` function. The argument "the solution is *advanced* because the number of tokens before the last `for` in the list comprehension is high and `zip` function is present." hit upon no counter examples. Moreover, no more critical or problematic examples were detected and thus the learning process concluded.

4.1 Assessment

During the interactive session student therefore also expressed his own (actually very sensible) suggestions on how to introduce new features into the learning domain or how to improve on the existing ones. Such new features can easily be incorporated into the teaching tool. Incidentally, the student's obtained rule model for determining whether a particular solution is advanced even outperformed the teacher in terms of classification accuracy on the testing data (90% vs. 83%; note that the same learning and testing set were used in all experiments). More importantly, the teacher found all given arguments and the obtained rules sensible.

The whole interactive procedure consisted of only 5 iterations and lasted about half an hour. This can be explained by the fact that the student only had to choose among the given attributes (and not yet to discover them). The student was selecting the attributes for his arguments rather skillfully, and in accordance with the given recommendations stated at the beginning of this section.

In the experiment with 7 students, the interactive learning session on average consisted of 7.1 iterations, the classification accuracy of students' final rule models on the testing data were (on average) 87.1% (AUC: 0.74, Brier: 0.25), while they themselves (on average) correctly classified 86.7% examples of the (previously unseen) testing data.

5 Conclusion

We demonstrated the use of argument-based machine learning (ABML) knowledge refinement loop [3, 7] for the purpose of knowledge elicitation of a difficult, ill-defined concept of distinguishing between "basic" and "advanced" programming style in python programming language, and used the results of knowledge elicitation for designing an interactive teaching tool. For the first time, ABML knowledge refinement loop was used in an interaction between a *student* and the computer. An interactive learning session between the student and the computer was thus described in detail. The initial experimental results with students are very promising, and suggest that ABML knowledge refinement loop provides a valuable interactive teaching tool. As a line of future work, we consider designing an online multi-domain learning platform based on student's argumentation of automatically selected examples and counter examples.

References

1. Ala-Mutka, K., Uimonen, T., Järvinen, H.M.: Supporting students in C++ programming courses with automatic program style assessment. JITE 3, 245–262 (2004)
2. Groznik, V., Guid, M., Sadikov, A., Možina, M., Georgiev, D., Kragelj, V., Ribarič, S., Pirtošek, Z., Bratko, I.: Elicitation of neurological knowledge with argument-based machine learning. Artificial Intelligence in Medicine 57(2), 133–144 (2013)
3. Guid, M., Možina, M., Groznik, V., Georgiev, D., Sadikov, A., Pirtošek, Z., Bratko, I.: ABML knowledge refinement loop: A case study. In: Foundations of Intelligent Systems. pp. 41–50. Springer Berlin Heidelberg (2012)
4. Li, N., Tian, Y., Cohen, W.W., Koedinger, K.R.: Integrating perceptual learning with external world knowledge in a simulated student. In: AIED. pp. 400–410 (2013)
5. Matsuda, N., Keiser, V., Raizada, R., Tu, A., Stylianides, G., Cohen, W., Koedinger, K.: Learning by teaching SimStudent: Technical accomplishments and an initial use with students. In: Aleven, V., Kay, J., Mostow, J. (eds.) Intelligent Tutoring Systems, pp. 317–326. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2010)
6. Možina, M., Guid, M., Krivec, J., Sadikov, A., Bratko, I.: Fighting knowledge acquisition bottleneck with Argument Based Machine Learning. In: The 18th European Conference on Artificial Intelligence (ECAI). pp. 234–238. Patras, Greece (2008)
7. Možina, M., Žabkar, J., Bratko, I.: Argument based machine learning. Artificial Intelligence 171(10/15), 922–937 (2007)
8. O'Keefe, R.: The Craft of Prolog. Logic Programming, The MIT Press (2009)