SyntaxBased Analysis of Programming

Concepts in Python

**Abstract.** Writing programs is essential to learning programming. Most programming courses encourage students to practice with lab and homework assignments. By analyzing solutions to these exercises teachers can discover mistakes and concepts students are struggling with, and use that knowledge to improve the course. Students however tend to submit many different programs even for simple exercises, making such analysis difficult. We propose using tree regular expressions to encode common patterns in programs. Based on these patterns we induce rules describing common approaches and mistakes for a given assignment. In this paper we present a case study of rulebased analysis for an introductory Python exercise. We show that our rules are easy to interpret, and can be learned from a relatively small set of programs.

**Keywords:** Learning programming Educational data analysis Error diagnosis Abstract syntax tree Tree regular expressions

# 1 Introduction

Providing feedback to students is among the most timeconsuming tasks when teaching programming. In large courses with hundreds of students, feedback is therefore often limited to automated program testing. While test cases can reliably determine whether a program is correct or not, they cannot easily be associated with specific errors in the code.

Several attempts have been made to automatically discover common errors in student programs [[1](#_bookmark1) [4](#_bookmark3)]. This would allow a teacher to annotate a representative subset of submissions with feedback messages, which could then be automatically propagated to similar programs. These techniques are used for instance by the OverCode tool to visualize variations in student programs [[5](#_bookmark4)].

We use *tree regular expressions* to specify important patterns in a programs abstract syntax tree (AST) while disregarding irrelevant parts. We have previously demonstrated this approach with Prolog programs [[6](#_bookmark5)]. Here we refine the AST patterns and show that they can be applied to Python a different programming paradigm with only a few modifications.

# 2 AST Patterns

We encode structural patterns in ASTs using tree regular expressions (TREs). In this work we consider (only) patterns describing child and sibling relations in an AST. We write them as Sexpressions, such as (a (b d **.** e $) c)*.* This expression matches any tree satisfying the following constraints:

* the root a has at least two children, b and c, adjacent and in that order; and
* the node b has three children: d, followed by any node, followed by e.

As in ordinary regular expressions, caret (^) and dollar sign ($) anchor a node to be respectively the first or last sibling, and a period (.) matches any node.

Using TREs we can encode interesting patterns in a program while disregarding irrelevant parts. Take for example the following, nearly correct Python function that prints the divisors of its argument *n*:

def divisors(n):

for d in range(1, n): if n % d == 0:

print(d)

The highlighted fragments correspond to the following patterns:

1. (Function (body (For (body If)))) and
2. (Function (name divisors) (args Var $)

(body (For (iter (Call (func range) (args  **.** Var $))))))*.*

The first TRE encodes a single path in the AST and describes the programs controlflow structure: Function For If [[4](#_bookmark3)]. The second TRE relates the argument in the definition of divisors to the last argument to range. This pattern shows a common mistake for this problem: range(1,n) will only generate values up to n1, so n will not be printed as its own divisor. A correct pattern would include the operator + on the AST path to n, indicating a call to range(1,n+1).

Patterns are extracted automatically from student programs. We first canonicalize [[2](#_bookmark2)] each program using code from ITAP[1](#_bookmark0). For each pattern we select a subset of nodes in the AST, then construct the TRE by walking the tree from each selected node to the root.

While pattern extraction is completely automated, we have manually defined the kinds of node subsets that are selected. After analyzing solutions to several programming problems, we decided to use the following kinds of patterns:

We found that patterns constructed from such node subsets are useful for discriminating between programs. Note that every constructed pattern refers to at most one variable. We used patterns to induce classification rules for predicting program correctness. The next section demonstrates that these rules are easy to interpret in terms of bugs and strategies for a given problem.

# 3 Rules: A Case Study

The goal of learning rules in this paper is to discover and explain common approaches and mistakes in student programs. We use a similar rule learner to the one described in [[6](#_bookmark5)], implemented within the Orange data mining library [[7](#_bookmark6)]. Each program is represented in the feature space of AST patterns described in the previous section. Based on test results each program is classified either as *correct* or *incorrect*. Rules are then learned to explain why a program is correct or incorrect. Rules for incorrect programs are called *nrules* and rules for correct programs are called *prules*. Patterns mentioned within the condition part of a rule can be used to analyze student programming. For example, patterns from a rule for incorrect programs are more often present in incorrect than in correct programs, therefore they likely correspond to errors.

This section describes several rules induced for the Fahrenheit to Celsius Python exercise, which reads a value from standard input and calculates the result. To solve this exercise, a student must ask the user to input a temperature, and print the result. A sample correct program is:

Students have submitted 1177 programs for this problem, with 495 correct and 682 incorrect programs. Our system extracted 891 relevant AST patterns, which were used as attributes in rule learning. The rule learner induced 24 nrules and 16 prules.

Two examples of highly accurate nrules were:

P20 incorrect [208, 1]

P5 P35 incorrect [72, 0]

The first rule covers programs where the pattern P20 is present. The rule implies an incorrect program, and covers 208 incorrect and one correct program. P20 is the AST pattern describing a call to the int function:

(Module (body (Assign (value (Call (func (Name (id int) (ctx Load))))))))

The pattern P5 in the second rule matches programs where the result of the input call is not cast to float but stored as a string. Pattern P35 matches programs where the value 32 is subtracted from a variable on the lefthand side of a multiplication. Sample programs matching the first rule (left) and the second rule (right) are:

These rules describe two common student errors. The left program fails when the user inputs a decimal. The right program is incorrect because the input string must be cast to a number. Not casting it (pattern P5) and then using it in an expression (pattern P35) will raise an exception.

In some cases, nrules imply a missing pattern. For example:

P0 incorrect [106, 0]

Pattern P0 matches programs with a call to function print. A program without a print is always incorrect, since it will not output anything.

Let us now examine the other type of rules. A sample prule was: P2 P8 correct [1, 200]

Patterns in the condition of the rule, P2 and P8, correspond respectively to expressions of the form float(input(?)) and print((?32)\*?). Programs matching both patterns wrap the function float around input, and have an expression that subtracts 32 and then uses multiplication within the print.

This rule demonstrates an important property of prules: although patterns P2 and P8 are in general not sufficient for a correct program (it is trivial to implement a matching but incorrect program), only one out of 201 student submissions matching these patterns was incorrect. This suggests that the conditions of prules represent critical elements of the solution. Once students have figured out these patterns, they are almost certain to have a correct solution. A sample program matching this rule.

# 4 Discussion and Further Work

Our primary interest in this paper is to help manual analysis of student submissions. We proposed to first represent submitted programs with patterns extracted from abstract syntax trees and then learn classification rules that distinguish between correct and incorrect programs. We showed that both rules and patterns are easy to interpret and can be used to explain typical mistakes and approaches.

The accuracy of automatic classification plays a secondary role, but it is a good measure to estimate the expressiveness of patterns. Over 12 exercises a random forest model achieved about 17% overall higher accuracy than the majority classifier. This result indicates that a significant amount of information can be gleaned from simple syntaxoriented analysis. To further improve the quality of patterns, we intend to analyze misclassified programs in exercises and derive new formats of patterns, which should enable better learning.

We have demonstrated how AST patterns can be encoded with TREs, and how patterns can be combined to discover important concepts and errors in student programs. Currently, analyzing patterns and rules is quite cumbersome. We plan on developing a tool to allow teachers to easily construct and refine patterns based on example programs. Ideally we would integrate our approach into an existing analysis tool such as OverCode [[5](#_bookmark4)].