Syntax-based analysis of programming concepts in Python

Martin Možina, Timotej Lazar

University of Ljubljana, Faculty of Computer and Information Science, Slovenia

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 content and presentation of 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 describe the rule-learning algorithm and present two case studies of rule-based analysis for introductory Python problems. 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 time-consuming 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.

Prompt, specific feedback could however greatly benefit learning. Furthermore, analyzing the problems students have with programming exercises can allow teachers to improve the course. The main obstacle to such analysis is the large variability of student submissions: even for the simplest tasks, students can submit tens of thousands of different programs [5,11].

Several attempts have been made to automatically discover commonalities in a set of programs [6,12,9,4]. 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 [3].

This paper presents a new language for describing patterns in student code. Our approach is based on *tree regular expressions* (TREs) used in natural language processing [8]. TREs are similar to ordinary regular expressions: they allow

us to specify important patterns in a program's abstract syntax tree (AST) while disregarding irrelevant parts. We found that TREs are sufficiently expressive to represent various concepts and errors in novice programs.

We have previously demonstrated this approach with Prolog programs [7]. Here we refine the definition of AST patterns, and show that they can be applied to Python – representing a different programming paradigm – with only a few language-specific modifications. We also demonstrate that rules learned from such patterns can be easily interpreted.

In CodeWebs, Nguyen et al. look for functionally equivalent AST subtrees that perform the same transformation from inputs to outputs [9]. Piech et al. learn program embeddings to a similar end [10]. Our work differs from theirs mainly in that TREs allow us to describe relations in (potentially disjoint) subparts of the program. We induce rules based on binary correct/incorrect labels, and do not have to execute any submission more than once.

Jin et al. use linkage graphs to represent dependencies between individual program statements based on the variables they contain [6]. This allows them to form equivalence classes of programs by ignoring unrelated statements. While TREs can encode such attributes, we use smaller patterns that encode dependencies between pairs of variables. Hovemeyer et al. focus on control structures such as loops and conditionals [4]. Our AST patterns include TREs that describe control flow, expressing the same features.

When comparing programs we must account for superficial differences such as different naming schemes. Rivers et al. canonicalize student programs using equivalency-preserving transformations (renaming variables, reordering binary expressions, inlining functions and so on) [12]. We use their canonicalization as a preprocessing step before extracting patterns.

The next section describes TREs and AST patterns, and gives a brief evaluation to show that patterns can discriminate between correct and incorrect programs. Section 3 describes our modified version of the CN2 rule-learning algorithm, and analyzes student solutions to two programming exercises in terms of such rules. The last section outlines the directions for our future work.

2 AST patterns

We encode structural patterns in ASTs using tree regular expressions (TREs). An ordinary regular expression describes the set of strings matching certain constraints; similarly, a TRE describes the set of trees containing certain nodes and relations. Since TREs describe structure, they are themselves represented as trees. More specifically, both ASTs and TREs are ordered rooted trees.

In this work we used TREs to encode (only) child and sibling relations in ASTs. We write them as S-expressions, such as `(a (b ^ d . e $) c)`. This expression matches any tree satisfying the following constraints (see Fig. 1 for an example):

- 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`.

Analogous to ordinary regular expressions, caret (`^`) and dollar sign (`$`) anchor a node to be respectively the first or last child of its parent. A period (`.`) is a wildcard that matches any node.

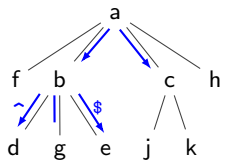

Fig. 1. A tree matching a pattern (blue arrows besides the edges). In the pattern, each arrow $x \to y$ means that node x has a child y. A shorter line without an arrowhead (e.g. `b— g`) indicates a wildcard, where the child can be any node. Anchors `^` and `$` mean that the pattern will match only the first or last child.

With TREs we 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)
```

Figure 2 shows the simplified AST for this program, with two patterns overlaid. These patterns are represented by the S-expressions

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 program's block structure: Function—For—If. The second TRE relates the argument in the definition of `divisors` with the last argument to `range` that provides the iterator in the for loop. Since S-expressions are not easy to read, we will instead represent TREs by highlighting relevant text in examples of matching programs:

```
def divisors(n):
    for d in range(1, n):
        if n % d == 0:
            print(d)
```

The second pattern shows a common mistake for this problem: `range(1,n)` will only generate values up to `n-1`, so `n` will not be printed as its own divisor. A correct pattern would include the binary operator `+` on the path to `n`, indicating a call to `range(1,n+1)`.

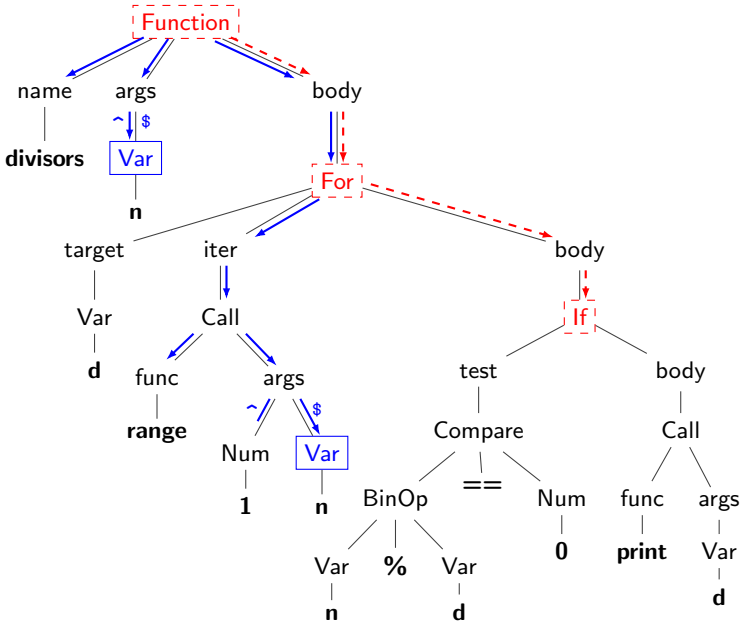

Fig. 2. The AST for the `divisors` program with two patterns. Leaf nodes (in bold) correspond to terminals in the program, i.e. names and values. Dashed red arrows represent the pattern describing the control structure of the program. Solid blue arrows encode the incorrect second argument to the `range` function.

2.1 Constructing patterns

Patterns are extracted automatically from student programs. We first canonicalize each program [12] using code from ITAP[1]. To construct TREs describing individual patterns, we select a subset of nodes in the AST, and walk the tree from each selected node to the root, including all nodes along those paths.

Depending on node type we also include some nodes adjacent to such paths. For each comparison and unary/binary expression on the path we include the corresponding operator. For function definitions and calls we include the function name. Finally, in all argument lists we include the anchors (`^` and `$`) and a wildcard (`.`) for each argument not on the path. This allows our TREs to discriminate between e.g. the first and second argument to a function.

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. Figure 2 shows two examples of the first two kinds of patterns.

1. We select each pair of leaf nodes referring to the same variable.

[1] Available at `https://github.com/krivers/ITAP-django`.

2. For each control-flow node n we construct a pattern from the set $\{n\}$; we do the same for each Call node representing a function call.
3. For each expression (such as `(F-32)*5/9`) we select the different combinations of literal and variable nodes in the expression. In these patterns we include at most one node referring to a variable.

Note that in every constructed pattern, all Var nodes refer to the same variable. We found that patterns constructed from such nodesets are useful for discriminating between programs. As we show in Sect. 4, they are also easily interpreted in terms of bugs and strategies for a given problem.

3 Learning rules

The goal of learning rules in this paper is to discover and explain common approaches and mistakes in student programs. We use a rule learner called ABCN2e, implemented within the Orange data mining library [2]. ABCN2e modifies the original CN2 algorithm [1] to learn unordered rules; modifications are described in a technical report at `https://ailab.si/abml`.

General rule-learning algorithms, such as CN2, tend to generate many specific rules. This produces more accurate results but makes rules harder to explain. This section describes the problem-specific configuration of the rule-learning algorithm for extracting relevant and explainable patterns from student programs.

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*. A program can be incorrect for one of two reasons: either a) it contains some incorrect pattern (a buggy pattern) that should be removed or modified, or b) it is missing one or more programing constructs (patterns) that should be present for the program to be correct.

Classification rules can express both reasons. For buggy patterns we learn rules for incorrect programs, where each condition in the rule must express the presence of a pattern. The condition of such a rule therefore contains a set of patterns that imply a bug in the program. For missing patterns, we learn another set of rules covering programs that are not covered by above rules. These rules may contain missing patterns within their conditions, and describe the missing constructs in a program that have to be implemented. All rules explaining incorrect programs are called *n-rules*.

To learn explainable, meaningful and non-redundant rules, we impose the following additional constraints on the rule learner:

- classification accuracy of each rule must exceed 90%, because we accept a 10% false-positive error as acceptable;
- each conjunct in the condition of a rule must be significant according to the likelihood test, meaning that each pattern in the condition part is indeed relevant (we set the significance threshold to p=0.05);
- a condition can have at most 3 patterns; and

– each rule must cover at least 5 distinct programs – this avoids redundant rules that represent the same error with a different combination of patterns.

Different approaches can be represented with rules explaining correct programs. A program is correct when it implements all required patterns and no buggy patterns. There may be several possible sets of required patterns for each exercise, with each set corresponding to a different approach to solving it.

We use the same constraints as in the case of n-rules and learn rules for correct programs called *p-rules*. In this case, we always require that conditions mention the presence of patterns, since it is easier to explain different approaches of students with something they have written and not with something they have not. To account for possible buggy patterns, the requirement to achieve 90% classification accuracy was not evaluated on full data, but only on data not covered by n-rules. Hence, a rule can cover an example with a specific approach even though it contains a buggy pattern.

4 Interpreting rules

Learned rules can be used to analyze student programing. This section describes several rules induced for two Python exercises: Fahrenheit to Celsius, which reads a value from standard input and calculates the result, and Greatest Absolutist, one of the introductory exercises for functions.

4.1 Fahrenheit to Celsius

The first problem in CodeQ Python course is to write a program converting from degrees Fahrenheit to degrees Celsius. The program should ask the user to input a temperature, and print the result. A sample correct program is:

```
F = float(input("Fahrenheit: "))
C = 5 / 9 * (F - 32)
print("Celsius: ", C)
```

Students have submitted 1177 programs for this problem, with 495 correct and 682 incorrect programs. Our systems extracted 891 relevant AST patterns, which were used as attributes in rule learning. The rule learner induced 24 n-rules, 14 of which mention only presence of patterns, and 16 p-rules.

We first take a look at n-rules that mention only presence of patterns in their conditions. The most accurate rule according to the rule learner was:

P20 ⇒ incorrect [208, 1]

This rule covers programs where the pattern P20 is present. It 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 second best n-rule covers 72 incorrect and no correct programs:

P5 ∧ P35 ⇒ incorrect [72, 0]

Pattern P5 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 left-hand side of a multiplication. Sample programs matching the first rule (left) and the second rule (right) are:

```
g2 = input()
g1 = int(g2)
print(((g1-32)*(5/9)))
```

```
g2 = input('Temperature [F]? ')
g1 = ((g2 - 32) * (5 / 9))
print(g2, 'F equals', g1, 'C')
```

These rules describe two common student errors. The left program is incorrect, since it 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.

The two most accurate n-rules with missing patterns in their conditions are:

¬P0 ⇒ incorrect [106, 0]
¬P1 ∧ P16 ⇒ incorrect [100, 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.

The second rule covers programs with P1 missing and P16 present. P16 matches programs with a call to the `print` function, where the argument contains a formula which subtracts 32 from a variable and then further multiplies the result. P1 describes a call to the function `float` as the first item in an expression, i.e. `= float(...)`. This rule therefore represents programs that have the formula in the `print` function (P16 is present), however fail to cast input from string to float (P1 is missing).

Let us now examine the other type of rules. The best four p-rules are:

P2 ∧ P8 ⇒ correct [1, 200]
P1 ∧ P42 ⇒ correct [0, 68]
P1 ∧ P8 ⇒ correct [3, 217]
P80 ⇒ correct [0, 38]

Patterns in the condition of the first 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 first rule demonstrates an important property of p-rules: 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 p-rules represent the critical elements of the solution. Once a student has figured out these patterns, they are almost certain to have a correct solution. A sample program matching the first rule is:

```
g1 = float(input('Temperature [F]: '))
print(((g1 - 32) * (5 / 9)))
```

The second and third p-rules are variations of the first. For instance, the second rule describes programs that have the formula in the argument to the `print` function. The fourth rule, however, is different. P80 describes programs that subtract 32 from a variable cast to float. The following program matches P80:

```
g1 = input('Fahrenheit?')
g0 = ((float(g1) - 32) * (5 / 9))
print(g0)
```

4.2 Greatest Absolutist

In this exercise students must implement a function that accepts a list of numbers and returns the element with the largest absolute value. One solution is

```
def max_abs(l):
    vmax = l[0]
    for v in l:
        if (abs(v) > abs(vmax)):
            vmax = v
    return vmax
```

We have received 155 submissions (57 correct, 98 incorrect) for this exercise. Due to its higher complexity and since the solutions are much more diverse, we obtained 8298 patterns to be used as attributes in learning. High number of patterns together with a low number of learning examples could present a problem for rule learning: since the space of possible rules is large, some of the learned rules might be a result of statistical anomalies. One needs to apply a certain amount of caution when interpreting these rules.

The rule-learning algorithm learned 15 n-rules (7 mentioning only presence of patterns) and 6 p-rules. Below we can see the two best n-rules referring to the presence of patterns and two programs; the left one is covered by the first rule, and the right one by the second rule:

P64 $\Rightarrow$ incorrect [22, 0]
P2 $\wedge$ P70 $\Rightarrow$ incorrect [17, 0]

```
def max_abs(l):
    vmax = 0
    for i in range(len(l)):
        if vmax < abs(l[i]):
            vmax = l[i]
    return vmax
```

```
def max_abs(l):
    vmax = None
    for v in l:
        if vmax==None or vmax<v:
            vmax = abs(v)
    return vmax
```

The pattern from the first rule, P64, matches functions returning the variable that is used in the condition of an if clause without an application of another function (such as `abs`). The left program demonstrates this pattern, where the value `vmax` is compared in the if clause and then returned. According to the teachers of the Python class, this error is common, because students forget that they need to compare the absolute value of `vmax`.

The second rule contains two patterns. P70 (blue) matches functions containing the call to `abs` in an assignment statement nested within a for loop and an if clause. P2 (red) matches functions that return the variable used in an assignment statement within a for-if block. Such programs are incorrect because they do not store the original list element. For example, if -7 has the largest absolute value in the list, then the function should return -7 and not 7.

The best two n-rules with absence of patterns in condition are:

$\mathsf{P1} \wedge \neg\mathsf{P11} \wedge \neg\mathsf{P131} \Rightarrow \mathsf{incorrect}$ [34, 0]
$\mathsf{P36} \wedge \neg\mathsf{P162} \Rightarrow \mathsf{incorrect}$ [26, 0]

The first rule covers programs matching P1 (checks for a function definition in the program) but missing P11 (if the iteration variable in a for loop is directly assigned to another variable within an if clause) and P131 (whether the return statement uses indexing, i.e. `return l[?]`). One such example is the above right program: it contains a function definition, it does not directly assign the value of `v` but uses its absolute value, and does not use indexing in the return statement.

This rule specifies two missing patterns, which makes is quite difficult to understand. It does not directly state the issue with a given program: if one of the two missing patterns were implemented, the rule would not cover this program any more. Therefore, the questions is, which of these two reasons is really missing? Different missing patterns could be understood as different options to finalize the program.

The second rule identifies only one missing pattern. P36 matches a call to `max` in the return statement, whereas P162 matches a call to `max` with the given list as the argument. This rule covers, for example, the following program:

```
def max_abs(l):
    return max(abs(l))
```

It uses `max` in the return statement, but does not apply it directly to the input list `l`. Note that this program would fail because the function `abs` does not accept a list argument.

The four most accurate p-rules induced by our rule learner were:

$\mathsf{P11} \wedge \mathsf{P17} \wedge \mathsf{P35} \Rightarrow \mathsf{correct}$ [0, 20]
$\mathsf{P11} \wedge \mathsf{P27} \wedge \mathsf{P3} \Rightarrow \mathsf{correct}$ [2, 34]
$\mathsf{P519} \Rightarrow \mathsf{correct}$ [0 9]
$\mathsf{P27} \Rightarrow \mathsf{correct}$ [6 38]

Sample programs covered by the first and second rules are:

```
def max_abs(l):
    vmax = 0
    for v in l:
        if abs(vmax) < abs(v):
            vmax = v
    return vmax
```

```
def max_abs(l):
    vmax = l[0]
    for v in l:
      if abs(vmax) < abs(v):
            vmax = v
    return vmax
```

The first two rules and the above programs are similar. Both rules share a common reason, P11 (blue in both programs), describing a pattern, where the variable from the for loop is used in the right side of an assignment within the if clause. P17 and P27 are also similar (red in both programs). The former links the occurrence of a variable within the `abs` function in an if condition with the variable from an assignment, whereas the latter links the same variable from an if condition with the variable from the return statement. P35 matches variable assignments to 0, hence the first rule covers solutions initializing `vmax` to zero. P3 matches for-looping over the input list.

After inspecting all covered examples of the first and the second rule, we found out that the first rule is only a more strict version of the second rule, since all examples covered by the first rule are also covered by the second rule. These two rules therefore do not describe two different approaches, but two different representations of the same approach. Similarly, the fourth rule is a generalization of the first two rules, containing only P27 within conditions. This pattern seem to be particularly important. Of 44 programs, where students used the absolute value of `vmax` in comparison and returned `vmax` at the end, 38 were evaluated as correct.

The third rule describes a different pattern. It covers programs that define a list containing values 2, 1, and -6. Defining such a list is evidently not necessary for the solution of this exercise. Why would it then correlate with the correctness of the solution?

To explain this rule we first have to describe how students test their programs. One option is to simply use the *Test* button, which submits the program to a server, where it is tested against a predefined set of test cases. The other option is to click the *Run* button, which runs the program and outputs the results. Those students who defined a list with values 2, 1, and -6 in their programs are most likely using the second option. They create their own test cases and then submit a program only when they are certain that it is correct. Since the description of the exercise includes a single test case with values 2, 1, and -6, most students use this list as the testing case.

On the other hand, given that the rule covers only nine programs, the probability that the rule is a statistical artifact is not negligible.

5 Evaluation and discussion

Evaluation was performed on a subset of exercises from an introductory Python course, implemented in the online programming environment CodeQ[2]. Table 1

[2] Available at `https://codeq.si`. Source under AGPL3+ at `https://codeq.si/code`.

Table 1. Solving statistics, classification accuracy, and coverage of rules for several introductory Python problems. The second column shows the number of users attempting the problem. Columns 3 and 4 show the number of all / correct submissions. The next two columns show the classification accuracy for the majority and random-forest classifiers. The last three columns show percentages of covered examples: columns n_p and n give covered incorrect programs (n-rules with presence of patterns and all n-rules), and column p gives the percentage of correct programs covered by p-rules.

		Submissions		**CA**		**Coverage**		
Problem	**Users**	Total	Correct	Maj	RF	n_p	n	p
fahrenheit_to_celsius	521	1177	495	0.579	0.933	0.708	0.935	0.867
ballistics	248	873	209	0.761	0.802	0.551	0.666	0.478
average	209	482	186	0.614	0.830	0.230	0.338	0.618
buy_five	294	476	292	0.613	0.828	0.234	0.489	0.719
competition	227	327	230	0.703	0.847	0.361	0.515	0.896
top_shop	142	476	133	0.721	0.758	0.399	0.802	0.444
minimax	74	163	57	0.650	0.644	0.462	0.745	0.298
checking_account	132	234	112	0.521	0.744	0.143	0.491	0.115
consumers_anon	65	170	53	0.688	0.800	0.376	0.880	0.623
greatest	70	142	83	0.585	0.859	0.492	0.746	0.880
greatest_abs	58	155	57	0.632	0.845	0.612	0.878	0.789
greatest_neg	62	195	71	0.636	0.815	0.621	0.960	0.718
Total / average	2102	4811	1978	0.642	0.809	0.432	0.704	0.620

shows the number of users attempting each problem, the number of all / correct submissions, the performance of majority and random-forest classifiers for predicting program correctness based on patterns, and percentages of covered programs by rules: n_p is the percentage of covered incorrect programs with n-rules that contain only presence of patterns, n is the percentage of covered incorrect programs with all n-rules, and p is the coverage of correct programs with p-rules. Classification accuracies were obtained using 10-fold cross validation.

Our primary interest in this paper is finding rules to help manual analysis of student submissions. The accuracy of automatic classification thus plays a secondary role to the interpretability of our model, however it is a good measure to estimate the expressiveness of patterns. We see that AST patterns increase classification accuracy for about 17% overall. This result indicates that a significant amount of information can be gleaned from simple syntax-oriented analysis. Exceptions are the ballistics and the minimax problems, where there was no visible improvement. In both exercises, the set of used patterns was insufficient to distinguish between incorrect and correct programs. To further improve the quality of patterns, we will analyze misclassified programs in those two exercises and derive new formats of patterns, which will enable better learning.

The coverages of particular types of rules are also promising. On average, we can explain 43.2% of incorrect programs with the presence of patterns. Considering all n-rules, we can explain over 70% of incorrect submissions, however these rules are somewhat harder to understand. Similarly, we can explain 62% of cor-

rect programs with p-rules. The remaining 38% of solutions tackle the problems in ways that cannot be veiled by out current patterns or there are not enough similar solutions to form a rule. This requires further analysis. For example, in the checking_account exercise the coverage of p-rules is only 11.5%.

We have demonstrated how AST patterns can be described with TREs, and how such 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 and rules based on example programs. Ideally we would integrate our approach into an existing analysis tool such as OverCode [3].

References

1. Clark, P., Boswell, R.: Rule induction with CN2: Some recent improvements. In: Machine Learning – EWSL-91. pp. 151–163. Berlin (1991)
2. 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.: Orange: Data mining toolbox in Python. Journal of Machine Learning Research 14, 2349–2353 (2013)
3. Glassman, E.L., Scott, J., Singh, R., Guo, P.J., Miller, R.C.: OverCode: Visualizing variation in student solutions to programming problems at scale. ACM Transactions on Computer-Human Interaction (TOCHI) 22(2), 7 (2015)
4. Hovemeyer, D., Hellas, A., Petersen, A., Spacco, J.: Control-flow-only abstract syntax trees for analyzing students' programming progress. In: Proc. 2016 ACM Conf. on International Computing Education Research. pp. 63–72. ACM (2016)
5. Huang, J., Piech, C., Nguyen, A., Guibas, L.: Syntactic and functional variability of a million code submissions in a machine learning MOOC. In: Proc. Workshops 16th Int'l Conf. Artificial Intelligence in Education (AIED 13). pp. 25–32 (2013)
6. Jin, W., Barnes, T., Stamper, J., Eagle, M.J., Johnson, M.W., Lehmann, L.: 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 (2012)
7. Lazar, T., Možina, M., Bratko, I.: Automatic extraction of ast patterns for debugging student programs. In: International Conference on Artificial Intelligence in Education. pp. 162–174. Springer (2017)
8. Levy, R., Andrew, G.: Tregex and tsurgeon: tools for querying and manipulating tree data structures. In: 5th Int'l. Conf. Language Resources and Evaluation (2006)
9. Nguyen, A., Piech, C., Huang, J., Guibas, L.: Codewebs: scalable homework search for massive open online programming courses. In: Proc. 23rd Int'l World Wide Web Conf. (WWW 14). pp. 491–502 (2014)
10. Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M., Guibas, L.: Learning program embeddings to propagate feedback on student code. arXiv preprint arXiv:1505.05969 (2015)
11. Piech, C., Sahami, M., Huang, J., Guibas, L.: Autonomously generating hints by inferring problem solving policies. In: Proc. 2nd ACM Conference on Learning @ Scale (L@S 2015). pp. 195–204. ACM (2015)
12. Rivers, K., Koedinger, K.R.: Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor. International Journal of Artificial Intelligence in Education pp. 1–28 (2015)