Automating Program Structure Classification

Will Crichton
wcrichto@cs.stanford.edu
Stanford University

Georgia Gabriela Sampaio
gsamp@stanford.edu
Stanford University

Pat Hanrahan
Stanford University

ABSTRACT

When students write programs, their program structure provides insight into their learning process. However, analyzing program structure by hand is a time-consuming endeavor. We show how machine learning methods can automatically classify programs into high-level structures that are relevant to CSE researchers. We evaluate three models on classifying student solutions to the Rainfall problem: nearest-neighbors over syntax tree distance, recurrent neural networks, and probabilistic grammars. We achieve a maximum 88% classification accuracy, and explore the trade-offs and failure cases of each model. Finally, we show how probabilistic grammars can be used as an human-interpretable theory of student problem solving to classify multiple program structures simultaneously.

KEYWORDS

Program structure classification, plan composition, machine learning, neural networks, probabilistic grammars

ACM Reference Format:
Will Crichton, Georgia Gabriela Sampaio, and Pat Hanrahan. 2020. Automating Program Structure Classification. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

When a teacher creates a new programming assignment, they often wonder: what different kinds of solutions did my students come up with? And what can I learn about my students from their solutions? For teachers and CSE researchers alike, classifying the *program structure* of student solutions can help identify misconceptions [22], success predictors [20], and problem solving milestones [23]. Studies on plan composition — how students combine code templates to solve programming problems — have long used program structure to analyze student problem solving. For example, Fisler's 2014 study of plan composition in functional languages showed that certain high-level program structures correlated with increased error rates [5].

However, analyzing program structure is challenging, time-intensive work. In personal correspondence, Fisler estimated that hand-coding program structure for her 2014 study took 1-2 minutes per program. This estimate is consistent with Wu et al. who reported that hand-labeling student misconceptions in Code.org programs took an average of 2 minutes per program [22]. For CS1 courses with hundreds of students, such a per-program cost is prohibitive, suggesting the use of automation to alleviate the labeling burden.

The fundamental challenge in automating program structure classification is balancing the quality of human expertise with the quantity of automated analysis. For example, several systems have been developed to automatically cluster student solutions based on their syntactic structure [6, 7]. However, clusters may not always map to meaningful categories. By virtue of being fully automated, clustering systems provide users with no means to provide feedback. Carefully shaping categories is critical for meaningful results — prior work like Wu et al. and Fisler both used domain expertise, not automated clustering, to identify salient categories to frame their analysis.

An ideal tool for classifying program structures should have the following qualities:

- **Domain-independent:** the tool should work for programs in different languages (e.g. Python, Java, OCaml, and C) and different domains (e.g. CS1, data science, graphics).
- **Teachable:** the tool should be able to learn from feedback (e.g. "this program is classified incorrectly") or knowledge (e.g. "these two functions mean the same thing") provided by the user.
- **Interpretable:** the tool should be able to explain why a program was classified a particular way to promote transparency, accountability, and debuggability.

In this paper, we explore a variety of modern machine learning methods that satisfy these qualities (to varying degrees). We seek to characterize how effectively these methods can classify the types of program structures that are relevant to CSE researchers. Specifically, we use Fisler's Rainfall programs and categories as a representative task for program structure classification. Our contributions are:

- Evaluating a nearest-neighbors classifier, a recurrent neural network, and a probabilistic grammar on Fisler's Rainfall dataset, achieving a peak 88% classification accuracy.
- Characterizing the trade-offs and failure cases for each method based on the type of solution structures, programming languages, and other factors.
- Demonstrating the use of a probabilistic grammar as an interpretable theory of student problem solving by classifying multiple kinds of program structure in one model.

2 RELATED WORK

2.1 Program classification

Many kinds of high-level program analysis can be viewed as program classification. Plagiarism detection systems like MOSS [3] take a given student's program, and classify other programs as "plagiarized" or "different". We do not consider this task as program

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
let rec help (alon : float list) =
  match alon with
  | [] -> []
  | hd::tl ->
    if hd = (-999.) then []
    else if hd >= 0. then hd :: (help tl)
    else help tl : float list)
let rec rainfall (alon : float list) =
  match alon with
  | [] -> failwith "no proper rainfall amounts"
  | _::_ ->
    (List.fold_right (+.) (help alon) 0.) /.
      (float_of_int (List.length (help alon))
```

```
let rec cut_list (a : int list) =
  match a with
  | [] -> []
  | hd::tl ->
    if hd <> (-999) then hd :: (cut_list tl) else []
let non_neg (a : int list) =
  List.filter (fun x -> x > 0) a
let average (a : int list) =
  (List.fold_right (fun x -> fun _val -> x + _val) a 0) /
    (List.length a)
let rainfall (a : int list) =
  average (non_neg (cut_list a))
```

Figure 1: Two example "Clean First" OCaml programs. Even for the same high-level structure, student solutions exhibit a significant diversity in syntactic and semantic variation such as the function decomposition strategies shown here.

structure classification, as plagiarism systems predominantly compare low-level syntax differences, e.g. whether two programs are the same modulo renamed variables.

Other prior works attempt to classify the kind of problem being addressed in a program, or what algorithm is being used. Taherkhani and Malmi [18] classify sorting algorithms from source code using decision-trees on hand-engineered problem-specific code features. For example, their features included "whether or not the algorithm implementation uses extra memory" and "whether from the two nested loops used in the implementation of the algorithm, the outer loop is incrementing and the inner decrementing." Moving away from hand-engineering program features, the software engineering community has also applied deep learning techniques for similar tasks. Mou et al. [13] defined a task of classifying which of 104 programming competition problems a program is attempting to solve, and they apply a novel tree-based neural network for this task. Bui et al. [4] introduce bilateral neural networks to solve the same problem in a language-independent manner. In this work, we focus on classifying how a student solved a problem, as opposed to what problem they were solving.

Closer to our application domain, Wu et al. [22] evaluate a recurrent neural network (RNN) and multimodal variational autoencoder on classifying misconceptions in Code.org programs. In their problem formulation, a student can have one of 20 misconceptions about geometric concepts, and the goal is to classify which misconceptions a student has from their program. Malik and Wu et al. [12] introduce a method for neural approximate parsing of probabilistic grammars, achieving human-level accuracy on misconception classification. We adapt the RNN and probabilistic grammar models in Sections 4.1.2 and 4.2.

2.2 Program clustering and similarity

In a classification problem, a fixed set of categories is given up front. The role of a model is to classify data into one of these predetermined categories. Clustering methods attempt to solve a more challenging problem by simultaneously discovering the categories and the mapping from data to category. In the education community, prior work in program clustering has used classical heuristics such as computing edit distance between abstract syntax trees [8] and control-flow ASTs [7], or finding exact matches on canonicalized source code [6] and control-flow graphs [11]. Edit distance has also been applied to program repair [2] and code clone detection [21]. While we do not try to solve the category discovery problem, we still capture the underlying method of these approaches with a syntax-tree edit-distance nearest-neighbors baseline.

Recent work has also applied machine learning techniques to learn program comparison metrics from data. Tufano et al. [19] use a recursive autoencoder on identifiers, syntax trees, control flow graphs, and bytecode to build a semantic embedding space for programs, then use embedding space distance for clone detection. Raychev et al. used decision trees [15] and conditional random fields [16] to learn associations between code fragments for predicting the values and types of holes in programs. While these metrics are likely more robust than tree edit distance, they are challenging to adapt for niche teaching languages like Pyret. For example, Raychev et al. used 150,000 JavaScript files to learn an embedding for JavaScript, and we strongly suspect we cannot find that much Pyret code out in the world.

3 DATASET

Our goal is to evaluate program structure classification methods on programs and classes relevant to CSE researchers, i.e. classifying strategies on student programs, not classifying the kind of problem solved in LeetCode solutions. We chose to replicate the hand-labeled program structures used in Fisler's study of plan composition in functional solutions to the Rainfall problem [5]. In that study, students were prompted with:

> Design a program called rainfall that consumes a list of numbers representing daily rainfall amounts as entered by a user. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list.

Fisler's dataset contains student solutions in three functional languages: OCaml, Pyret, and Racket. Across these languages, Fisler identified three high-level structures ("Single Loop", "Clean First",

and "Clean Multiple") that accounted for a large majority of student solutions. Each category indicates a different choice of when and how to filter the input list for valid rainfall data. Single Loop fuses summing/counting with filtering, Clean First filters the list then sums and counts the clean data, and Clean Multiple separately filters in the summing and counting logic. Figure 1 shows examples of two Clean First OCaml programs, and Section 4 of Fisler's paper contains further discussion.

Overall, the dataset consists of 136 OCaml and 42 Pyret student solutions to the Rainfall problem. Each solution's structure has been hand-labeled by a human expert (either Fisler or the current authors). In Section 4, we describe the methods for automatically classifying Rainfall program structures, and in Section 5 we evaluate the methods on Fisler's dataset.

4 METHODS

We selected machine learning methods to evaluate in accordance with the criteria proposed in the introduction: **domain-independent**, **teachable**, and **interpretable**. Focusing on teachable methods rules out unsupervised solutions like program clustering, as they generally do not accept feedback beyond hand-tuning program features. We considered two kinds of teachable methods:

- **Supervised learning:** the standard way to "teach" a machine learning classifier is through training data. In this framing, a teacher would hand-label a number of student programs with their respective category, and then train a classifier on the labeled data. Providing feedback is simply labeling more training data.
- **Programmable theories:** an intriguing new approach is to use an explicit model of student reasoning, e.g. a probabilistic grammar, to analyze student programs. This allows a teacher to more directly express their domain knowledge than through labeled training data.

First, we discuss the supervised methods (nearest-neighbors and RNN), then the programmable theory method (probabilistic grammar). We address the structure of each method, and its domain-independence and interpretability.

4.1 Supervised learning

Within the category of supervised learning, the main design decisions are: what is the representation of the input program to the method? And how does the method learn from training data? We select two approaches, nearest-neighbors and RNN, with different trade-offs of interpretability, domain-independence, and accuracy.

4.1.1 Nearest-neighbors. A nearest-neighbors classifier represents a simple baseline for supervised program structure classification, as it has a simple formulation, requires no training, and makes interpretable decisions. To explain, let's set up the mathematical structure of the problem. The input is a dataset $\mathcal{D} = \{(p_1, l_1), \ldots, (p_n, l_n)\}$ of n pairs of programs p and labels l. For example, the two programs in Figure 1 both have l = "Clean First".

Given a new program p', a nearest-neighbors classifier finds the most similar program p_i from the training data, and assigns p' the label l_i. Formally, given a distance function:

$$\text{Dist} : \text{Program} \times \text{Program} \to \mathbb{R}$$

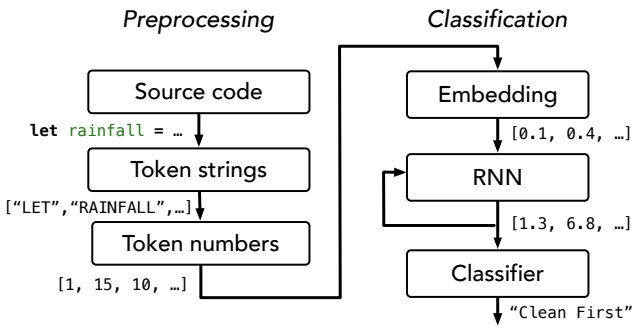

Figure 2: RNN classifier architecture. Programs are converted into integer sequences where each number uniquely identifies a token. Each token is mapped to a k_e dimensional embedding, fed through the RNN to produce a k_h dimensional hidden state. At the sequence end, a classifier predicts the plan label from the hidden state.

A nearest-neighbors classifier classifies a new program p' as:

$$\underset{p, l \in \mathcal{D}}{\text{argmin}}\, \text{Dist}(p, p')$$

This method is interpretable in that the classifier can directly produce the nearest training program p as a justification for its classification.

The key design decision is choosing a distance metric. One metric that has been widely used for program similarity is *tree edit distance*. When two programs are represented as their abstract syntax tree (AST), the edit distance is the number of tree manipulation operations needed to transform one tree into the other. For our classifier, we use the canonical Zhang-Shasha method [24].

Similar to prior work [7], we do not compare syntax trees verbatim. Small syntactic differences like choice of variable name or presence of type annotation do not usually impact the structure of a program. For both OCaml and Pyret, we use their respective compilers to generate a raw AST, then erase variable names, constant values, and type annotations before computing edit distance.

4.1.2 Recurrent neural network. A downside to nearest-neighbors is that the distance metric is a hand-engineered feature, susceptible to issues where programs may be syntactically similar but structurally different (or vice versa, as in Figure 1). Neural networks are a supervised learning method that automatically learn features from the training data. In practice, learned features can increase accuracy (with enough training data), but decrease interpretability. We use a *recurrent* neural network because our input programs do not have a fixed size, unlike e.g. convolutional neural networks for image classification.

We adapt a basic RNN architecture from Wu et al. [22] as shown in Figure 2. Each token of the source program is mapped to a high-dimensional embedding before being processed by the RNN. (Although a token is represented as a number, tokens are still categorical, not ordinal data. For example, if `let` = 1, `fun` = 2, `end` = 10, the network shouldn't learn that "`let`" and "`fun`" are somehow more related than "`end`" by virtue of being assigned closer identifiers.) An RNN processes the sequences of token embeddings to

```
helper_in_body = random_choice([True, False])
return format("""
    {%- if not helper_in_body -%}
    {{helper_body}}
    {%- endif -%}

    let {{recursion}} rainfall {{params}}
        {%- if helper_in_body -%}
        {{helper_body}} in
        {{rainfall_body}}
        {% else -%}
        {{rainfall_body}}
        {% endif -%}
    """, helper_in_body, ...)
```

Figure 3: An example random choice from the OCaml grammar implemented in Python with Jinja2 formatting. Based on the `helper_in_body` random variable, a program is generated with helper functions either in or out of the rainfall function.

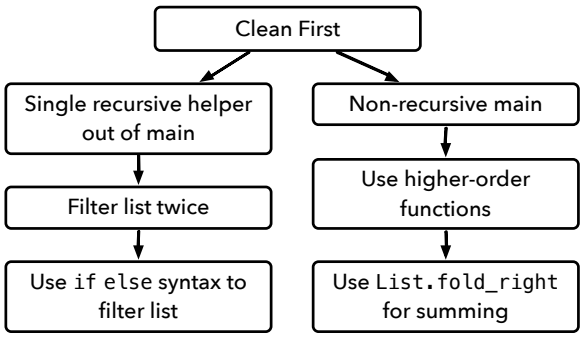

Figure 4: The simplified approximate parse tree using the OCaml grammar for the program in Figure 1-left. Decisions are nested, e.g. if the student used higher-order functions, then we case on whether they use `fold_left` or `fold_right`.

produce a hidden state vector. A logistic regression layer then classifies the hidden state into a probability distribution over the class labels.

We use standard optimization techniques [10] to train the RNN model on labeled student data. We use the model weights from the training iteration with highest accuracy on a validation set as the version for evaluation. Hyperparameters such as the embedding dimension size are selected by automated search over the parameter space.

4.2 Programmable theories

Supervised learning methods are "teachable" in the sense that they can be provided more training data. A model makes a prediction, a person says the prediction is incorrect, and a black box training algorithm attempts to learn from the mistake. However, this kind of feedback is fairly primitive, because one cannot convey *why* the correct classification is as it is. The models do not enable domain expertise to be easily expressed.

An alternative approach is to use a *programmable theory* (or *generative model* in ML lingo). An expert writes a "student simulator" that embodies a theory of reasoning by which a student would create a program. Such a simulator can incorporate an expert's knowledge about problem solving and program structure. Then to solve a classification problem, the simulator is run in reverse to determine: what is the most likely simulation that would have generated a given student program?

Programmable theories have a long history in production systems and cognitive tutors [1], where the goal was to model problem-solving so as to interactively teach students high-level programming skills. Production systems have not been historically useful in modeling realistic human behavior. However, new models that integrate machine learning techniques have shown promise in bridging the gap between domain expertise and statistical inference.

4.2.1 Probabilistic grammars. In this work, we adopt the approach introduced by Wu et al. [22] of using probabilistic grammars to model student reasoning. A grammar has two key components: first, a series of decisions a student could make in generating a program, like "uses a Clean First structure", "computes list length using a helper function", and so on. Second, a grammar describes how each decision becomes a program.

For example, imagine we ask students to write a program to compute $x = y * 2$. If we observe two programs `let x = y * 2` and `let x = y + y`, we can generalize them into a grammar:

```
binop_style = sample(['add', 'mult'])
expr = 'y * 2' if binop_style == 'mult' else 'y + y'
return f'let x = {expr}'
```

This Python program is a grammar that can generate the two observed student programs. The `binop_style` variable represents a decision in the grammar, i.e. a part of our theory of how students generate programs. The `sample` function makes the grammar probabilistic — when run in the forward direction, this programmable theory will randomly generate a student program.

For the Rainfall dataset, we constructed probabilistic grammars for OCaml and Pyret by observing a training set of student programs, and iteratively building a grammar that could generate each program. At each step, we applied our programmer's intuition to identify the salient decisions in how students structured their programs. Figure 3 shows an excerpt from the OCaml probabilistic grammar, and Figure 4 shows an example decision tree as parsed from the program in Figure 1-left.

4.2.2 Neural approximate parsing. While a probabilistic grammar can be run forward to randomly generate a program, our goal is to run it in reverse: given an actual student program, what are the most likely decisions that would have generated it? This is a classic parsing problem, of matching a grammar to a string.

However, unlike traditional parsing, we assume that many student programs may not be precisely generated by the grammar. In our earlier example of multiplying by 2, if we wanted to parse `let x = 2 * y`, our grammar cannot exactly parse this string, but we would consider it "closer" to a multiplication style than an addition style. Hence, we want to perform an *approximate* parse, i.e.

to find the closest program in the grammar to the actual student program.

We adopt the technique described in Malik et al. [12] for *neural approximate parsing* (NAP), i.e. using a neural network to perform the approximate parse. At a high level, the probabilistic grammar is used to randomly generate thousands of training examples of synthetic student programs paired with the decisions made by the simulator. A neural network is trained with this data to approximately parse programs under the grammar. Rather than just classifying the program structure as with the supervised models, the parser attempts to determine every decision in the grammar.

4.2.3 Tradeoffs and auxiliary classification. In terms of our method criteria, a significant downside to this approach is a lack of domain-independence. Each grammar was laboriously hand-crafted for each language. We believe however that, given enough experience, many of the basic choices represented in the grammar (e.g. is a helper function inside or outside the main function) could be factored into a core library of program transformations that could be reused across grammars.

By contrast, a key upside is this method's interpretability. When used to classify programs, the grammar outputs not just the final classification, but also all the intermediate decisions that led to the classification. This additional information can be used for other kinds of analyses. For example, in Fisler's rainfall study, the author correlated program structure with a variety of other hand-labeled metadata about the student programs, such as where the counting logic was located. From her coding manual:

> Use "own" if plan is in separate function primarily designed for that plan, such as a separate sum function which may also handle negs and sentinel. Use "helper" if plan is one of many computations in a function other than "rainfall".

By this rule, the example in Figure 1-left is "rainfall" and Figure 1-right is "helper."

We hypothesized that the probabilistic grammar's intermediate decisions could be useful for classifying these kinds of auxiliary metadata. To test this hypothesis, we wrote the heuristic shown in Figure 5 to classify the counting logic location. Although the grammar was not created with the counting location properties in mind, several of the grammar's decisions provided sufficient information to classify the location correctly (if the parse was correct). If the heuristic works, then it supports the more general hypothesis that explicit, programmable theories of program generation are useful for flexibly addressing many interesting questions about student behavior.

```
def countwhere_heuristic(parse):
  if parse.plan == Plan.CleanFirst:
    return (CountWhere.Rainfall
      if parse.average_strategy == 'inline'
      else CountWhere.Helper)
  elif parse.plan == Plan.CleanMultiple:
    return (CountWhere.Rainfall
      if parse.anonymous_helpers
      else CountWhere.Own)
  elif parse.plan == Plan.SingleLoop:
    return (CountWhere.Helper
      if parse.body_spec in ['direct_pass', 'match']
      else CountWhere.Rainfall)
```

Figure 5: Heuristic function for classifying where counting logic occurs in a student's solution based on the inferred structure from the approximate parse.

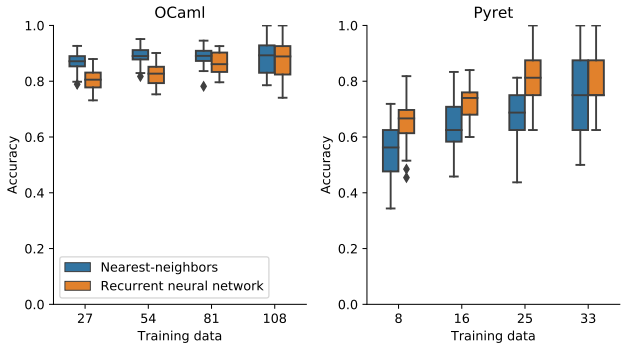

Figure 6: Distribution of model accuracy for different sizes of training and test sets and under each language. Each experimental condition is computed through 30 trials, so its distribution is visualized as box plot.

5 EVALUATION

The primary question of the evaluation is: how accurate is each model in classifying Rainfall program structures on Fisler's dataset? We answer this question under a variety of conditions to understand the generality of the models.

5.1 Supervised learning models

We start by evaluating the nearest-neighbors and RNN models (collectively the supervised models) as they share the same interface of training directly on labeled student data. For program structure classification with supervised models, we are interested in two kinds of generality:

- Generalizing to unseen data: if I train a model on k programs, how likely will it classify the $k + 1^{\text{th}}$ program correctly?
- Generalizing to other tasks: how does model accuracy change as the program structure and the programming languages change?

5.1.1 Model generality. To evaluate the supervised models, we train our models on a portion of labeled student data, and evaluate the accuracy of the models on a separate held-out test set. Because the datasets are small, rather than privileging some particular subset as the canonical test set, every experiment is run as a k-fold Monte Carlo cross validation. We select a proportion p of the overall dataset to be training data and the rest to be testing data, then train and evaluate on k different random partitions. This methodology produces a distribution of model accuracies. Essentially, we run a series of simulations of the question "if I started randomly labeling student data, what is the probable range of accuracy the

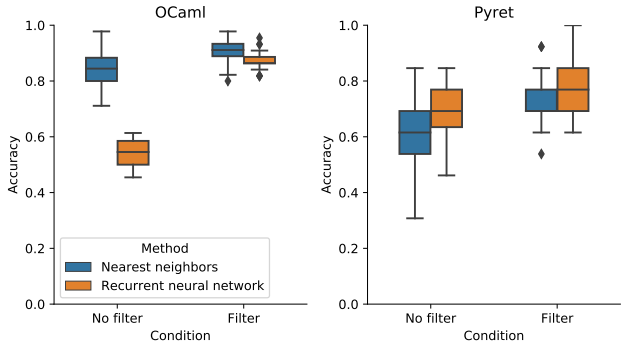

Figure 7: Effect of filtering tests from programs on classification accuracy.

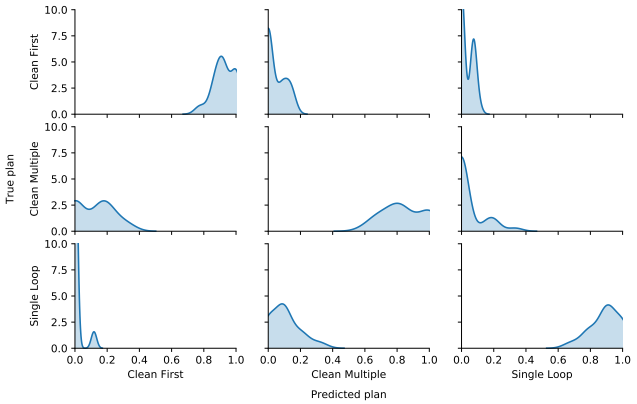

Figure 8: Distribution of row-normalized confusion matrices for the RNN method in OCaml. The x-axis of each subplot shows the probability P(Predicted plan | True plan), e.g. the upper right plot shows the probability a program is classified as Single Loop given its true plan is Clean First. The y-axis shows the number of times a confusion matrix had a given probability across the 30 folds. Kernel density estimation is used to smooth the empirical distribution of probabilities.

model would have for a given amount of training data?" For all experiments, we use $k = 30$ folds.

Figure 6 compares the nearest-neighbors and RNN models on both languages across $p = 20\%$, 40%, 60%, and 80% training set proportions. Overall, the results suggest that the RNN method can consistently generalize well with similar or less variance than nearest-neighbors when provided sufficient training data. A few key observations:

- The highest mean accuracy in OCaml is 88% for both the nearest-neighbors and RNN models when training on 108 data points. For Pyret, the highest mean is 72% for nearest-neighbors and 81% for RNN.
- Nearest neighbors outperforms RNN under every training set size for OCaml, and the converse is true for Pyret.
- The mean standard deviation for both models in OCaml was $\sigma = 0.042$, while it was $\sigma = 0.100$ for Pyret.

For nearest-neighbors, the performance gap between the two languages is likely explained by AST size. While the average program length in tokens is 116 for OCaml vs. 127 for Pyret, the average program size in number of AST nodes is 50 for OCaml and 196 for Pyret. This difference is likely an artifact of the implementation of ASTs in the respective compilers. Further work in simplifying the AST could potentially improve nearest-neighbors performance.

5.1.2 Filtering tests. To reduce spurious variance in the student solutions, we remove all unit tests from the source programs. In OCaml, we eliminate every top-level statement that is not a function definition. In Pyret, we eliminate every statement in a `where` block. Both methods start with the output of this preprocessing step, e.g. with programs like those shown in Figure 1.

To evaluate the impact of filtering unit tests from the student solutions, we perform a cross-validation of each model type for both languages under an 70/30 train/test split. In the first condition we do not filter tests, tokenizing/parsing the student solutions as provided to us, and the second condition applies the filtering described above.

Results are shown in Figure 7. For OCaml, the RNN method gets 25% additional percentage points in accuracy from filtering tests. For Pyret, gains are more modest, but variance diminishes significantly. Overall, these results suggest that reducing spurious variance in the student data can have a large impact on accuracy.

5.1.3 Error analysis. To further understand the limitations of our methods, we analyze their failure modes. We perform a cross validation with a 70/30 train/test split. We then visualize the distributions of errors in the confusion matrix shown in Figure 8. The error distribution for nearest-neighbors looks approximately the same as the RNN, so we only show the RNN plot.

The plot shows that true Clean First programs are misclassified less often and equally between the other two classes. Clean Multiple has a greater misclassification rate, being most frequently confused with Clean First. And Single Loop is almost exclusively misclassified as Clean Multiple, a somewhat confusing asymmetry given Clean Multiple is rarely misclassified as Single Loop.

While the RNN does not have any obvious ways to understand its decision-making, an advantage of the nearest-neighbors is that each classification comes with an interpretable explanation: the closest program. Hence, we can dig into the nearest-neighbor misclassifications to see how syntactic similarity is mistaken for plan equivalence, as shown in Figure 9. While the two programs are only one example, they demonstrate how syntactic similarity through edit distance can struggle to capture the true nuance of distinguishing plans.

5.2 Probabilistic grammar

The probabilistic grammar approach requires a different evaluation as it involves more human intervention than the supervised models. We treat any student data that the human observes in creating the grammar as training data, even if the approximate parser is not trained directly on the student data. Specifically, we randomly selected 1/3 of the labeled student OCaml programs (45 in total) for the grammar writer (one of the authors) to observe while writing and testing the grammar. The author attempted to write a grammar

```
let rec rainfall_help1 (alon : int list) =
  match alon with
   | [] -> 0
   | (-999)::tl -> 0
   | hd::tl -> hd + (rainfall_help1 tl)
let rec rainfall_help2 (alon : int list) =
  match alon with
   | [] -> 0
   | (-999)::tl -> 0
   | hd::tl -> 1 + (rainfall_help2 tl)
let rainfall (alon : int list) =
  (float_of_int (rainfall_help1 alon)) /.
     (float_of_int (rainfall_help2 alon))
```

```
let rec positive lst =
  match lst with
  | [] -> []
  | head::tail ->
    (match head with
      | (-999) -> []
      | x when x > 0 -> head :: (positive tail)
      | x when x < 0 -> positive tail)
let rec sum_list lst =
  match lst with
  | [] -> 0
  | head::tail -> head + (sum_list tail)
let rec rainfall lst =
  (sum_list (positive lst)) / (List.length (positive lst))
```

Figure 9: The left program is a Clean Multiple solution that was misclassified by the nearest-neighbors classifier as Clean First, being matched with the Clean First program in the training set on the right. The functions share significant syntactic and structural similarity, e.g. two helper functions, similar style of match, and similar top-level usage. However, the helper functions are critically used in very different ways.

Method	Nearest-neighbors	RNN	NAP
Accuracy	0.88	0.85	0.84

Table 1: Plan classification accuracy of each model on the human-unobserved student data.

program that captured all of the structural and syntactic variation amongst the 45 programs, e.g. making rules like the one in Figure 3. The final grammar was a 425-line Python program.

Using the grammar, we generated 1,000 unique Rainfall solutions and trained the neural approximate parser on the synthesized data. Our main goals for evaluation are to understand: how well does the model generalize to the human-unobserved data, and how can the interpretability of the model be used for auxiliary classification tasks?

5.2.1 Model generality. To evaluate the accuracy and generality of the NAP classifier, we compare the classification accuracy of NAP on the 92 unobserved OCaml programs against the discriminative models trained directly on the 45 observed data points. The results are shown in Table 1. For the particular train/test split in this experiment, the NAP classifier gets comparable accuracy to the other two classifiers. This result suggests that neither the additional synthetic training data nor the more difficult parsing task meaningfully increase accuracy over supervised methods for Rainfall program structure classification.

5.2.2 Auxiliary classification. Using the heuristic shown in Figure 5, we computed the count logic location as described in Section 4.2.3. We then compared the heuristically classified labels to the manual labels in the dataset.

The heuristic gets 83% accuracy on all 136 OCaml programs. A brief analysis of the errors showed that most failure cases were due to an incorrect parse (i.e. wrong structure prediction) as opposed to a failure of the heuristic logic. This suggests that as the quality of the grammar and approximate parsing models improves, these kinds of auxiliary classification tasks will become easier to solve.

6 DISCUSSION

Our evaluation demonstrates that these methods can achieve high accuracy for program structure classification of problems at the Rainfall level of complexity. Going forward, we hope that these tools will enable researchers to scale up program structure analysis to many programs with high accuracy. Understanding the structure of student code at scale can potentially inform the design of CS education curricula, such as the conclusions reached in Fisler's study [5].

Moreover, as the CSE community pursues theories of student learning and problem solving with a renewed vigor [9, 14], tasks like program structure classification offer a means for evaluating the usefulness of mechanized theory. Quoting Simon and Newell [17], "the programmability of the theories is the guarantor of their operationality, an iron-clad insurance against admitting magical entities into the head." While we did not explicitly apply any learning theory in the creation of our grammar, we think that programmable theories like probabilistic grammars provide a useful platform for utilizing knowledge about reasoning processes.

Potential avenues for future work include developing program structure classification methods for more complex and hierarchical program structures. and using automation to aid the plan discovery process. Across these application areas, we hope that a mixture of human expertise and data-driven insight can help program structure classification provide a deeper, empirically validated perspective on how students learn to program.

7 ACKNOWLEDGEMENTS

We are deeply grateful to Kathi Fisler for providing us with her Rainfall dataset used in this paper.

REFERENCES

[1] John R Anderson, Robert Farrell, and Ron Sauers. 1984. Learning to program in LISP. *Cognitive Science* 8, 2 (1984), 87–129.
[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *arXiv preprint arXiv:1902.06111* (2019).

[3] Kevin W Bowyer and Lawrence O Hall. 1999. Experience using" MOSS" to detect cheating on programming assignments. In *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011*, Vol. 3. IEEE, 13B3–18.

[4] Bui Nghi DQ, Yijun Yu, and Lingxiao Jiang. 2019. Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 422–433.

[5] Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the tenth annual conference on International computing education research*. ACM, 35–42.

[6] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 7.

[7] David Hovemeyer, Arto Hellas, Andrew Petersen, and Jaime Spacco. 2016. Control-flow-only abstract syntax trees for analyzing students' programming progress. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 63–72.

[8] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*, Vol. 25.

[9] Yasmin Kafai, Chris Proctor, and Debora Lui. 2019. From Theory Bias to Theory Dialogue: Embracing Cognitive, Situated, and Critical Framings of Computational Thinking in K-12 CS Education. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. ACM, 101–109.

[10] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[11] Andrew Luxton-Reilly, Paul Denny, Diana Kirk, Ewan Tempero, and Se-Young Yu. 2013. On the differences between correct student solutions. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. ACM, 177–182.

[12] Ali Malik, Mike Wu, Vrinda Vasavada, Jinpeng Song, John Mitchell, Noah Goodman, and Chris Piech. 2019. Generative Grading: Neural Approximate Parsing for Automated Student Feedback. *arXiv preprint arXiv:1905.09916* (2019).

[13] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[14] Greg L Nelson and Andrew J Ko. 2018. On Use of Theory in Computing Education Research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 31–39.

[15] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 731–747.

[16] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 111–124.

[17] Herbert A Simon and Allen Newell. 1971. Human problem solving: The state of the theory in 1970. *American Psychologist* 26, 2 (1971), 145.

[18] Ahmad Taherkhani and Lauri Malmi. 2013. Beacon-and schema-based method for recognizing algorithms from students' source code. *Journal of Educational Data Mining* 5, 2 (2013), 69–101.

[19] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 542–553.

[20] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. 2017. Learning to Represent Student Knowledge on Programming Exercises Using Deep Learning.. In *EDM*.

[21] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. 2018. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1066–1077.

[22] Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. 2019. Zero shot learning for code education: Rubric sampling with deep learning inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 782–790.

[23] Lisa Yan, Nick McKeown, and Chris Piech. 2019. The PyramidSnapshot Challenge: Understanding student process from visual output of programs. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 119–125.

[24] Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.