Comparing Feature Engineering Approaches to Predict Complex Programming Behaviors

Wengran Wang, Yudong Rao, Yang Shi, Alexandra Milliken, Chris Martens,
Tiffany Barnes, Thomas W. Price
North Carolina State University
{wwang33, yrao3, yshi26, aamilllik, crmartens, tmbarnes, twprice}@ncsu.edu

ABSTRACT

Using machine learning to classify student code has many applications in computer science education, such as autograding, identifying struggling students from their code, and propagating feedback to address particular misconceptions. However, a fundamental challenge of using machine learning for code classification is how to represent program code as a vector to be processed by modern learning algorithms. A piece of programming code is structurally represented by an abstract syntax tree (AST), and a variety of approaches have been proposed to extract features from these ASTs to use in learning algorithms, but no work has directly compared their effectiveness. In this paper, we do so by comparing three different feature engineering approaches for classifying the behavior of novices' open-ended programming projects according to expert labels. In order to evaluate the effectiveness of these feature engineering approaches, we hand-labeled a dataset of novice programs from the Scratch repository to indicate the presence of five complex, game-related programming behaviors. We compared these feature engineering approaches by evaluating their classification effectiveness. Our results show that the three approaches perform similarly across different target labels. However, we also find evidence that all approaches led to overfitting, suggesting the need for future research to select and reduce code features, which may reveal advantages in more complex feature engineering approaches.

1. INTRODUCTION

Automatically classifying student code using machine learning has many applications in computer science education, such as to automatically grade students' code [8], to predict when students are unlikely to succeed at a task and may benefit from feedback [13], and to propagate feedback on particular misconceptions to students who need it [10]. However, a fundamental challenge in applying machine learning to source code is how to represent that code in a way the learner can understand. The structure of programming code is traditionally represented as an abstract syntax tree (AST), where nodes and their children correspond to specific code elements (e.g., an if statement), and the tree can be arbitrarily large. However, the vast majority of machine learning models take fixed-length vectors as input.

In many domains, researchers have addressed this challenge of code representation by extracting a set of *features* from source code, which can then represent the code in the model. For example, a simple Bag-of-Words (BoW) approach represents code as a binary vector, where each element indicates the presence or absence of a specific AST node anywhere in the code (e.g., [6, 18, 11]). However, simple feature extraction approaches like BoW do not capture the complex structural relationships among AST nodes in student code, which may be important for many classification tasks. A number of other feature extraction approaches have been proposed (e.g., [2, 28, 4]), but no work has directly compared their effectiveness for classifying student code. Further, feature extraction can be especially difficult in the domain of computer science education, where students' code may cover a large and sparse solution space, with little overlap among solutions paths [21, 27, 13]. This suggests the need to develop new feature extraction approaches that address this challenge.

In this paper, we compared the effectiveness of three code feature extraction approaches, on a challenging and generalizable classification task. This task classifies programming game design projects to identify the presence and absence of complex game behaviors. We found that the three code feature extraction approaches had similar performance across all behaviors, and the performance of specific feature extraction approaches is dependent on factors such as the properties of the target label, the size of training data, and the prevalence of positive labels. Our work contributes to educational data mining for CS education by comparing the affordance of different feature engineering approaches and evaluating their effectiveness in predicting the presence of complex game behaviors.

2. RELATED WORK

In this section, first, we discuss the relevance and importance of automatic code classification for improving computing education through personalization and scalability. We then summarize state-of-the-art feature engineering approaches that related work has used for various code analysis purposes.

2.1 Applications of Classifying Student Code

Manual labeling of student code is a frequent practice in computing education. It is often done by instructors and researchers, for example, to grade student program submissions, identify misconceptions [15, 23], or to profile a programming dataset to identify when particular code features are used [12]. However, labeling tasks are quite time-consuming and hard to scale, leading many researchers to investigate methods for automatically labeling code.

Researchers have used different approaches to automatically classify and analyze students' code, such as using correct program submissions to generate rubric-based auto-graders [8], or using programming homework grades to infer students' knowledge [2]. Elmadami et al., for example, built a data-driven misconception classifier in EER-Tutor, an Intelligent Tutoring System that provides tutorials for database design. Using association rule mining, EER-Tutor categorizes frequent failing patterns as indicators of misconceptions [9]. Similarly, Mao et al. developed a classifier that predicts a student's success in completing a programming task based on their programming code trajectory with just one minute of data from a student [13]. This classifier, if implemented in programming education, could help instructors or learning systems to prompt students with suggestions or feedback when they most need it. In addition, our prior work has shown the efficacy of adding an automatic code classifier to a learning system. We developed an unsupervised classifier to identify completions of 11 sub-goals in a Snap*!* block-based programming task [27]. We then integrated this classifier in a programming environment to detect sub-goal completions and provide timely positive feedback to students, which significantly increased the time students spent engaged with the programming task [14]. These results suggest that, in programming learning environments, automatic code classification can help provide adaptive and scalable feedback to support students.

2.2 AST Structural Feature Extraction

Researchers have used various approaches to extract a fixed set of features from code to use in machine learning models. *Structural feature extraction* looks for patterns in an AST and creates a binary input vector, indicating the presence or absence of these patterns, or counts of the frequency of their occurrence. For example, Bag-of-Words (BoW) is a common feature extraction approach, adapted from natural language processing, where each possible type of AST node becomes a binary feature. Figure 1 shows how BoW features transform a piece of code into an input vector by indicating the presence or absence of each feature in the programming code AST. The BoW approach has been used in various code classification tasks, such as to predict students' success in completing a program, or to summarize functions of code snippets [6, 18, 11]. For example, Azcona et al. used BoW to represent students' code, and found that after using BoW feature extraction to convert program code into vectors, a simple Naive Bayes model predicted correctness of short pieces of Python code submissions with 59.4% accuracy [6]. This suggests that even a relatively simple feature extraction approach, such as BoW, extracts useful information that can predict meaningful labels for student code with some success.

BoW features represent a single AST node, regardless of its neighbors. However, meaningful programming patterns usually include nodes that are structurally connected with each other in the AST. For example, in Figure 2(b), the piece of programming code completes a behavior that, when a sprite [1] touches a bullet, the game ends. In order to accomplish this behavior, the "Stop" block must be inside of the "if" block - otherwise, the behavior would be different. In order to extract structural information - such as the requirement that the "stop" block be inside the "if" for the behavior (shown in Figure 2), more complex features can be extracted. For example, researchers have extracted features corresponding to *paths* within the AST. For example, the root path for a given node consists of the path from that node to the root node [21], and this has been used for data-driven hint generation [21]. The Code2Vec algorithm [4] also decomposes the AST into a collection of paths for use in a deep neural network (discussed further in Section 2.3), which was used to predict method names of code from Java GitHub repositories (not student code). Others have used n-Grams, which are n-length sequences of nodes extracted from a flattened representation of the AST. For example, a vertical n-Gram is created by a depth-first iteration of its nodes, and a horizontal n-Gram is created by a breadth-first iteration of all children in an AST subtree (shown in Figure 2(c)). Akram et al. used code n-Grams as features to predict the rubric-based grades of students' block-based programs with a Gaussian Process model that achieved an R-squared of 0.94, higher than the 0.88 achieved by the baseline BoW approach [2].

Many of these structural features can be represented more generally as a type of pq-Grams. A pq-Gram is a subtree that includes a target node, along with its $(p-1)$ ancestor nodes, as well as q of its child nodes. For example, Figure 2(b) shows how a pq-Gram can be extracted for the target node "script (2)", with p ancestor nodes and q child nodes. If a node has fewer than $(p-1)$ ancestors or q children, the pq-Gram includes this information by noting these missing nodes as "null". pq-Grams were introduced as part of a method to calculate differences between tree-structured data [5], such as between a student's AST and a correct solution's AST, in order to provide automated hints [28]. Using this notation, we can also consider the features extracted by the BoW and root path approaches to be pq-Grams: $p(1)q(0)$-Grams and $p(\infty)q(0)$-Grams, respectively. Most horizontal n-Grams can be represented as $p(0)q(n)$-Grams, and vertical n-Gram can be viewed as sub-arrays of root paths, and can be represented as $p(n)q(0)$-Grams. These AST structural feature extractions have been shown to be effective for representing and analyzing student code [27]. Despite the variety of feature extraction approaches, no work has compared the efficacy of these approaches, especially pq-Gram feature extraction, which has not been used previously for code classification tasks. This comparison is important, since having features with too *little* expressivity (e.g., BoW) will not capture important AST structural information (causing the resulting model to underfit), but having features with too *much* expressivity (e.g., treating an entire AST as a feature) will lead to overly-specific features that

[1] A sprite in Scratch is similar to an object-oriented class. In Scratch game design projects, an actor of the game is usually represented by a sprite.

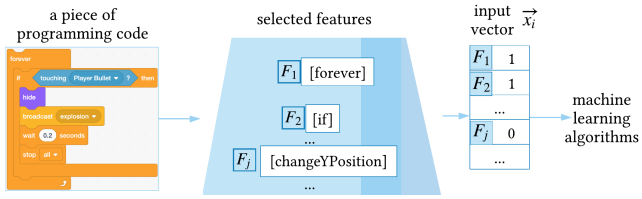

Figure 1: Using features selected from a set of programming code, such as the Bag-of-Words features in this figure, we can convert a piece of code into an input vector by indicating the count or presence of selected features.

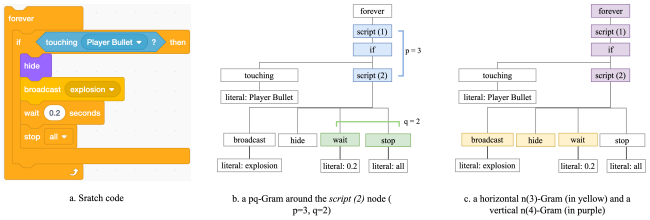

Figure 2: A piece of Scratch code can be represented as an AST. A pq-Gram centered on a node n includes n's $p-1$ ancestor nodes and up to q of n's child nodes, while an n-Gram includes a sequence of nodes inside an AST subtree, with length n, either horizontally or vertically.

do not generalize to new, unseen instances, causing the resulting model to overfit. Empirical evaluation is needed to find an appropriate balance.

Researchers have also explored clustering these simple features to represent more complex structural relationships. For example, Zhi et al. automatically clustered pq-Grams into what they called "features" [27], which clusters pq-Grams that performs a meaningful programming sub-goal. However, this clustering method was not applied to feature engineering for supervised classification. Mao et al. used Recent Temporal Patterns (RTPs) to transform a highly condensed feature set into a Multivariate State Sequence, including information such as feature co-occurrence and precedence. They used this feature engineering approach to predict whether a student is unlikely to succeed in a given task. Using RTPs, their model performs better than simple feature extraction approaches, with an 18.8% increase in classification accuracy, showing that creating feature combinations offers more information on students' programming status [13]. However, these existing approaches were only applied in short programming tasks with specific goals.

2.3 Distributed Code Embeddings

So far, we have discussed ways to represent code as one-hot or count vector. Neural Network models, in addition, commonly contain an embedding layer that learns a multi-dimensional representation on top of the feature extraction approaches we have discussed. In addition, many embedded approaches use sequential models, so that the embedding is trained based on relative locations of each code element [11, 18]. These embeddings can be learned in an unsupervised manner [11], similar to how word embeddings such as Word2Vec [16] are learned. They can also be learned in a supervised manner through back-propagation during model training [4, 18]. Depending on the architecture of Neural Network, these embedded approaches also vary in ways to represent code as vectors to feed into the models. For example, Alon et al. used a leaf-to-leaf approach, connecting the shortest path from each two leaf nodes [4, 3]. Iyer et al. used embedded Bag-of-Words code representations to feed into an LSTM model for conducting code retrieval tasks, and have shown that their model can answer programming questions by finding highly-relevant code snippets [11].

Although distributed code representation is a powerful way to interpret programs, they typically require a large amount of training data - Code2Vec, for example, was trained on more than 14M pieces of programming code. This requirement on the number of training samples is not suitable for our task.

3. EXPERIMENT AND RESULTS

Our goal in this study was to compare the effectiveness of the existing code feature extraction approaches outlined in Section 2.2. To do so, we compared the effectiveness of BoW, n-Gram, and pq-Gram approaches in classifying game behaviors.

3.1 Dataset and Classification Task: Labeling Open-Ended Scratch Projects

The student code used in our evaluation comes from the Scratch community [22], an online, novice-friendly, block-based programming website, where users create and remix interactive programming projects, such as games and animations. We chose the Scratch repository because it includes diverse, open-ended programs from learners around the world, which are not constrained to a single assignment or goal. This might be analogous to submissions to an open-ended final programming course project. On Feb 18th, 2020, we scraped the 6247 most trendy projects, from the *Game* genre in the Scratch community. Among them, we selected the first 457 projects based on the creation date. Among the 457 projects, we excluded 44 projects that had over 50%

broken or unused code [2]. Our dataset for the classification task includes 413 projects, with an average of 1201 AST nodes in each project [3].

An important classification task for Scratch game projects is to identify whether a given project includes a specific game behavior (i.e., a game mechanic), for example, whether the player can jump (like in the classic Mario game). Algorithm 1 shows one pseudocode example of how a platformer jump can be implemented in Scratch [4]. In this example, this behavior is implemented by two threads, to ensure that the actor jumps using gravity, and stops when landing on the top of a platform. Based on our observations of students' code, *PlatformerJump* is the most complex behavior in the five behaviors, usually including a large amount of code, spread across different sprites and scripts. However, even with a relatively less complex behavior (e.g., *CollisionChangeVar*), students can still implement the behavior in a wide variety of ways.

The ability to detect these game behaviors automatically would allow researchers to better understand novice programming behavior by profiling the whole Scratch repository, including millions of projects, to find popular combinations of behaviors (e.g., in [1]). It would also enable researchers to instantly identify what type of game a student is currently working on, in order to offer them highly customized feedback or examples. This task also represents a difficult challenge for code classification, since these game behaviors are comprised of many code elements, which may be dispersed throughout a student's code (e.g., the jump behavior, shown in Algorithm 1), and which may be implemented in diverse ways, creating a large and sparse programming state-space [27]. These challenging properties are shared by many other programming code classification tasks, such as identifying misconceptions [9] and predicting learner performance [13].

In order to create meaningful categories of game behaviors, the first author investigated 13 student game design project submissions, from an undergraduate programming course in a large, public research university. After thoroughly examining the submissions, the first author decomposed each game into a set of discrete game behaviors, under the criteria that these behaviors are general enough to be reused in other games. We identified 24 game behaviors. From these, we selected five that represented a diverse range of complexity and frequency of use. One author developed a definition for each behavior label (see Table 1) and trained another author on how to label projects, during which they jointly labeled 20 projects with the presence or absence of these five game behaviors. The two authors then individually labeled the rest of the game design projects.

[2]Many Scratch game design projects come from remixing and reusing existing projects [24], and novices do not always have the ability to effectively modify these projects [19, 20], leading to abandoned or broken designs [20].

[3]In block-based languages such as Scratch, an AST node generally has a corresponding block that matches the node. So this also means that we have an average of about 1201 blocks in each project.

[4]Algorithm 1 is an example implementation of the *PlatformerJump* behavior. The exact game logic is not important, but it illustrates the complexity of Scratch game behaviors.

Algorithm 1: PlatformerJump Example Pseudocode

```
1  begin
2      run in parallel
3          initialize position
4          velocity ← 0
5          forever
6              change y position by velocity
7          end
8      run in parallel
9          forever
10             while touching platform color do
11                 velocity ← 1
12             velocity ← −1
13             if up arrow key is pressed then
14                 velocity ← 3
15                 while not touching platform color do
16                     velocity ← velocity − 0.1
17                 velocity ← 0
18         end
```

Table 1 shows the description and the commonness of the behaviors in the 413 Scratch game projects. These behaviors have the following characteristics:

1. These behaviors are implemented by a variety of blocks, sometimes more than 30 blocks across different sprites (i.e., similar to object-oriented classes) or code scripts (i.e., threads).

2. Students implemented these behaviors in a variety of different ways, using varying types and numbers of blocks. This makes expert-authored rule-based static analysis (e.g., [25, 7]) ineffective at detecting the presence of these behaviors.

3. Although the selected game behaviors are typical within certain game genres, the prevalence of individual behaviors is often quite infrequent, as is shown by the counts of projects in Table 1, with a range of 5.3% to 46%. This creates imbalanced datasets, which pose a challenge in training classifiers that can lead a model to be biased towards the majority class.

While the above characteristics make our classification task challenging, they are also common characteristics in many important code classification tasks. Code indicating misconceptions, low performers, or notable strategies may also be complex, diverse, or rare. The results of our evaluation, therefore, may be able to generalize to these tasks as well.

3.2 Experiment Setup

In order to understand how well these feature extraction approaches capture meaningful information for predicting the presence of complex game behaviors, we compared the BoW, n-Gram, and pq-Gram approaches. Here we present our experiment setup.

Table 1: Target Labels of Game Behaviors

Label Name Abbreviation	Label description	# of projects with this label	# of blocks (estimate)
KeyboardMove	An actor [5] moves in the direction indicated by the player on the keyboard	197/413	3 - 10
CollisionChangeVar	When one actor touches another, a variable changes (e.g. score)	146 /413	4 - 6
PlatformerJump	An actor can jump and then falls down with gravity	81/413	20 - 50
MoveWithMouse	An actor moves when the user moves or clicks the mouse	49/413	2 - 4
CollisionStopGame	The game ends when an actor touches another	25/413	3 - 4

Feature extraction. We first extracted BoW, n-Gram, and pq-Gram features from the training dataset. To reduce the number of irrelevant features, we extracted features that have more than 5% support (i.e., the percentage of projects where this feature exists). When extracting n-Grams of a specific n value, we consider both horizontal and vertical ones, as introduced in Section 2.2, for they each extract different AST structural information. We extracted n-Grams with $n \in \{1, 2, \ldots, 10\}$, and also extracted pq-Grams, with $p \in \{1, 2, 3\}$, and $q \in \{1, 2, 3, 4\}$. The exact subset of these n-grams or pq-Grams used was determined by hyperparameters, as discussed below. At each increase of n in n-Grams, we kept features that were extracted by smaller ns, but removed duplicated smaller features based on the rule that when two features always co-appear, and one is a subset of another. We extracted pq-Grams using the same approach.

Training and evaluation. To train our model, we used a Support Vector Machine (SVM) model with a linear kernel, and used the regularization parameter as a hyperparameter, with values in {0.01, 0.1, 1, 10, 100}. We employed five-fold cross-validation to evaluate our feature set. Within each round of cross-validation, we used $\frac{1}{4}$ of the training set as the validation set to tune the hyper-parameters. When we extracted features for n-Grams, we used the maximum n as a hyperparameter, with $n \in \{1, 2, \ldots, 10\}$. Similarly, we also used the maximum ps and qs as hyperparameters when extracting pq-Grams, with $p \in \{1, 2, 3\}$, and $q \in \{1, 2, 3, 4\}$. The values of the hyperparameters were determined by their F1 scores on the validation set at each round of cross-validation. Since many of our target labels are highly imbalanced, the accuracy score offers little information on how well our model performs in predicting target labels. We therefore use F1 scores to tune hyperparameters.

3.3 Results

Figure 3 shows how the feature sets perform across the five target game behaviors. We present the F1 scores of prediction in each target behavior, with its precision (P) and recall (R), shown in the brackets. In this first experiment, our results show that all classifiers perform similarly. One exception to this is the *PlatformerJump* behavior, where BoW (F1 = 0.58) does notably worse than n-Gram (F1 = 0.72), or $pq-Gram$ (F1 = 0.68). We note that *PlatformerJump* is easily the most complex behavior (shown in Algorithm 1), which can be completed in many ways. This suggests that there may be some advantage to more expressive feature representations for identifying more complex program prop-

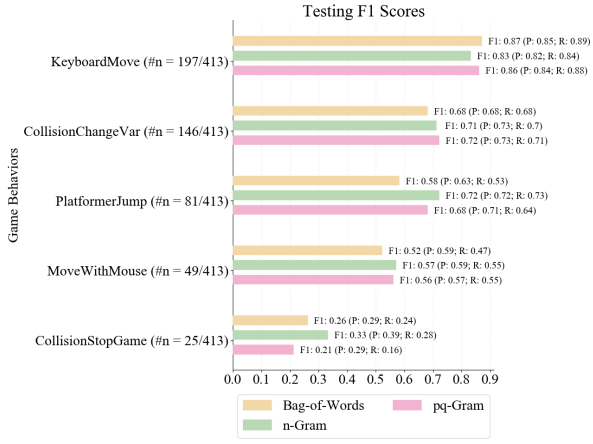

Figure 3: Comparing BoW, n-Gram, and pq-Gram approaches, our results show that these approaches alone achieve relatively similar predictive outcomes, and that F1 scores are lower when the prevalence of positive samples is relatively small.

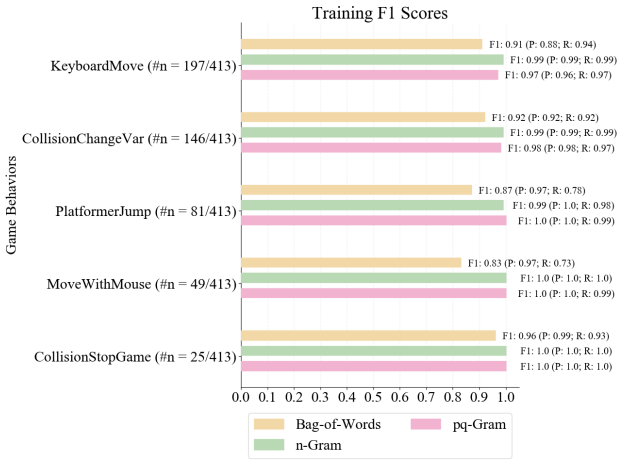

Figure 4: Near perfect training F1 scores suggest that the model may be overfitting with all three feature extraction approaches, especially among n-Gram and pq-Grams.

erties.

Figure 3 also shows that F1 scores of all approaches decrease as the prevalence of positive samples decreases (i.e., with more class imbalance). For example, on the y-axis of Figure 3, we have marked each label with the prevalence of its positive samples. As the prevalence of features decreases from 197/413 (48%) in *KeyboardMove* to 25/413 (6%) in *CollisionStopGame*, The feature extraction methods perform increasingly worse. For more common behaviors such as *KeyboardMove*, all approaches had sufficient data to accurately identify the behavior (F1 = 0.83-0.87), and even simpler approaches such as BoW were expressive enough to find discriminating features, e.g., the "WhenKeyPressed" block. However, when less positive training data are available, none of the approaches perform well, suggesting the possibility that the models are overfitting.

We therefore investigated the training F1 scores for each model, shown in Figure 4. The results confirm that the models are likely overfitting with all three feature extraction approaches, especially when the prevalence of positive samples is small, such as in *MoveWithMouse* and *CollisionStopGame*. This is unsurprising, given that all feature extraction approaches produced hundreds (BoW) to over a thousand (*pq*-Gram) features, and the training data never exceeded 200 positive instances. Because even our simplest feature extraction approach (BoW) was clearly overfitting, it is unclear whether more expressive feature representations (n-Grams, *pq*-Grams) would hold an advantage under other circumstances (e.g., more training data, with additional feature selection[6]).

4. DISCUSSION AND CONCLUSION

We have compared different approaches for extracting predictive features from the program code. We have also presented the first step towards automated classification of open-ended block-based program behaviors, going beyond existing rule-based analysis [17, 1]. Here we discuss the insights we have derived from our results, and we discuss our future task to improve our current classification approach and address underlying challenges.

Overall, we found no evidence that different feature extraction approaches led to better or worse code classification results. However, we note that in all cases, this was due to overfitting, due to a large number of features and a relatively small amount of training data. This is supported by the fact that all approaches did worse for behaviors with more class imbalance. We are therefore unable to conclude whether our results would generalize to a larger dataset, where the richer feature of n-Gram or *pq*-Grams may be better leveraged. However, we also note that in the domain of computer science education, courses often only have a relatively small number of students, causing the size of training data to be also small. This means that in programming code feature extraction approaches, we should explore ways to reduce the number of features, through feature selection and dimensionality reduction. It may also be helpful to develop ways to more efficiently label data - which serves as the building block for many intelligent algorithms.

[6]We did attempt basic feature selection approaches, reducing the number of features to less than 100, but this either failed to improve or worsened performance, suggesting a need for future work exploring how to address the overfitting.

Our classification performance also varied considerably across tasks, in some cases quite well (e.g. *KeyboardMove*), but in others quite poorly (e.g. *CollisionStopGame*). To understand why, we note that, unlike prior work, our code classification task used a relatively small dataset ($n = 413$), consisting of large programming projects. The Scratch projects we analyzed had an average of 1201 AST nodes. By contrast, code classification tasks in prior work have generally been applied to smaller code input, with a much larger amount of training data. For example, Code2Vec is implemented in a code classification task for predicting method names from programs with an average length of 7 lines [4], although the training sample is of size 14M. Iyer et al. implemented an LSTM-based code summarization model, but only on programming code with an average of 38 tokens (i.e., the number of elements in the program). Specifically, in the computer science education domain, many programming code analysis approaches are evaluated on short programming tasks, such as drawing a geometric shape using nested loops [26, 13, 27], or implementing a short algorithm (e.g., a bubble sort algorithm [2]). Mou et al. evaluated their LSTM-based code classifier's performance on classifying function methods, and concluded that longer programs (i.e., with longer length of code) had relatively lower classification performance compared to shorter programming tasks [18]. This may explain why our results were not as strong, and more prone to overfitting.

In conclusion, in this work, we compared features with different levels of expressivity (i.e., Bag-of-Words, n-Grams, and *pq*-Grams), in a challenging task to classify meaningful game design behaviors in open-ended Scratch projects. Our results show that our model may be overfitting with all three different feature extraction approaches, and that we need to explore ways to reduce feature dimensions and increase data size to improve performance.

5. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1917885.

6. REFERENCES

[1] E. Aivaloglou and F. Hermans. How kids code and how we know: An exploratory study on the scratch repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 53–61, 2016.

[2] B. Akram et al. Assessment of students' computer science focal knowledge, skills, and abilities in game-based learning environments. 2019.

[3] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A general path-based representation for predicting program properties. In *ACM SIGPLAN Notices*, volume 53, pages 404–419. ACM, 2018.

[4] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[5] N. Augsten, M. H. Böhlen, and J. Gamper. Approximate matching of hierarchical data using pq-grams. In *VLDB*, volume 5, pages 301–312, 2005.

[6] D. Azcona, P. Arora, I.-H. Hsiao, and A. Smeaton. user2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, pages 86–95. ACM, 2019.

[7] M. Ball. Lambda: An autograder for snap. Technical report, Technical Report. Electrical Engineering and Computer Sciences University of California at Berkeley, 2018.

[8] N. Diana, M. Eagle, J. Stamper, S. Grover, M. Bienkowski, and S. Basu. Data-driven generation of rubric criteria from an educational programming environment. In *Proceedings of the 8th International Conference on Learning Analytics and Knowledge*, LAK '18, page 16–20, New York, NY, USA, 2018. Association for Computing Machinery.

[9] M. Elmadani, M. Mathews, and A. Mitrovic. Data-driven misconception discovery in constraint-based intelligent tutoring systems. 2012.

[10] L. Gusukuma, A. C. Bart, D. Kafura, and J. Ernst. Misconception-driven feedback: Results from an experimental study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 160–168. ACM, 2018.

[11] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[12] C. M. Lewis. The importance of students' attention to program state: a case study of debugging behavior. In *Proceedings of the ninth annual international conference on International computing education research*, pages 127–134, 2012.

[13] Y. Mao, R. Zhi, F. Khoshnevisan, T. W. Price, T. Barnes, and M. Chi. One minute is enough: Early prediction of student success and event-level difficulty during novice programming tasks. In *EDM*, 2019.

[14] S. Marwan, T. W. Price, M. Chi, and T. Barnes. Immediate data-driven positive feedback increases engagement on programming homework for novices. 2020.

[15] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of programming in scratch. *ITiCSE'11 - Proceedings of the 16th Annual Conference on Innovation and Technology in Computer Science*, pages 168–172, 2011.

[16] T. Mikolov, K. Chen, G. Corrado, J. Dean, L. Sutskever, and G. Zweig. word2vec. *URL https://code. google. com/p/word2vec*, 2013.

[17] J. Moreno-León, G. Robles, and M. Román-González. Dr. scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia*, (46):1–23, 2015.

[18] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[19] G. L. Nelson, B. Xie, and A. J. Ko. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in cs1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 2–11. ACM, 2017.

[20] S. Papert. Children, computers and powerful ideas, 1990.

[21] T. W. Price, Y. Dong, and T. Barnes. Generating data-driven hints for open-ended programming. *International Educational Data Mining Society*, 2016.

[22] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[23] T. Sirkiä and J. Sorva. Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, pages 19–28, 2012.

[24] V. Vasudevan, Y. Kafai, and L. Yang. Make, wear, play: remix designs of wearable controllers for scratch games by middle school youth. In *Proceedings of the 14th international conference on interaction design and children*, pages 339–342, 2015.

[25] W. Wang, R. Zhi, A. Milliken, N. Lytle, and T. Price. Crescendo: Engaging students to self-paced programming practices. In *To be published in the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*, 2020.

[26] M. Wu, M. Mosse, N. Goodman, and C. Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 782–790, 2019.

[27] R. Zhi, T. W. Price, N. Lytle, Y. Dong, and T. Barnes. Reducing the state space of programming problems through data-driven feature detection. In *Educational Data Mining in Computer Science Education (CSEDM) Workshop@ EDM*, 2018.

[28] K. Zimmerman and C. R. Rupakheti. An automated framework for recommending program elements to novices (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 283–288. IEEE, 2015.