More With Less: Exploring How to Use Deep Learning Effectively through Semi-supervised Learning for Automatic Bug Detection in Student Code

Yang Shi*, Ye Mao*, Tiffany Barnes, Min Chi, Thomas W. Price
North Carolina State University
Raleigh, NC, USA
{yshi26, ymao4, tmbarnes, mchi, twprice}@ncsu.edu

ABSTRACT

Automatically detecting bugs in student program code is critical to enable formative feedback to help students pinpoint errors and resolve them. Deep learning models especially code2vec and ASTNN have shown great success for *large-scale* code classification. It is not clear, however, whether they can be effectively used for bug detection when the amount of labeled data is limited. In this work, we investigated the effectiveness of code2vec and ASTNN against classic machine learning models by varying the amount of labeled data from 1% up to 100%. With a few exceptions, the two deep learning models outperform the classic models. More interestingly, our results showed that when the amount of labeled data is small, code2vec is more effective, while ASTNN is more effective with more training data; for both code2vec and ASTNN, the more labeled data, the better. To further improve their effectiveness, we investigated the potential of semi-supervised learning which can leverage a large amount of unlabeled data to improve their performance. Our results showed that semi-supervised learning is indeed beneficial especially for ASTNN.

Keywords

CS Education, Machine learning, Program analysis, Bug detection, semi-supervised learning

1. INTRODUCTION AND BACKGROUND

When students encounter difficulties during programming, they are often caused by systemic procedural errors, or "bugs" [9], which can occur repeatedly across problems [38, 8]. For example, a student may confuse when it is appropriate to use the `and` and `or` operators, or fail to consider a boundary case in a condition, using `>` instead of `>=` [17]. These bugs are rarely directly addressed by the compiler or test-case feedback employed in most computer science (CS) courses, which are generally limited to suggesting syntax errors, or which correct input-output pairs the program fails to replicate. Historically, tutoring systems in a variety of learning domains have detected these bugs automatically (e.g. through a *bug library* [9, 4]). The detection can be used to offer tailored formative feedback [34] that address bugs directly [22], and can also help instructors to be more informed about student learning process [25]. The detection of bugs often requires experts' manual definitions, with distinct rules for detecting the bug on different problems [4]. This can make it impractical to use bug detection in practice. Most current automatic grading systems for student code are mainly based on test cases, which provide a score and failed test case information to students [15, 16, 37]. Nevertheless, the relationship between code's output and the presence of specific bugs in student code is not clear, since a given erroneous output could be caused by various errors in student code. An automatic bug detection system for student code could be useful to fill in the gaps for students.

Machine learning (ML) algorithms are powerful tools for data analysis, which have been commonly used for automatic programming code analysis [10]. Classical machine learning methods, such as support vector machines [13] and XGBoost [11], are capable of classifying program code [12, 21, 18]. Recent advances in machine learning have leveraged structural information in code to accurately classify and label it [2, 3, 41, 28]. For example, Alon et al. explored path representations on code represented as trees [2], and designed the code2vec model to learn the representations using deep neural networks [3]. Abstract Syntax Tree based Neural Network (ASTNN) by Zhang et al. applied recursive neural networks in the structure, outperforming Tree-based Convolutional Neural Networks [28] and other state-of-the-art models [41].

However, to apply the models to detect student program bugs, two challenges need to be addressed. First, these deep models were originally designed for *professional programs* which are fundamentally different than code written by students [39]. Some recent work has applied these techniques to educational domains [33, 19, 26, 30, 6], but they either used base models years before, [28, 19], or are not specifically used for bug detection [33, 26, 30, 6]. Second, deep learning models are traditionally "data hungry" [1], using large, labeled

*The first two authors contributed to the manuscript equally.

Yang Shi, Ye Mao, Tiffany Barnes, Min Chi and Thomas Price "More With Less: Exploring How to Use Deep Learning Effectively through Semi-supervised Learning for Automatic Bug Detection in Student Code". 2021. In: Proceedings of The 14th International Conference on Educational Data Mining (EDM21). International Educational Data Mining Society, 446-453. https://educationaldatamining.org/edm2021/
EDM '21 June 29 - July 02 2021, Paris, France

training datasets (e.g. [19] was trained on 270k samples). However, in most educational settings, datasets can be much smaller (e.g. ∼100 students), and labeling (e.g. to identify bugs) can take extensive expert effort [14]. This suggests the potential of leveraging a *semi-supervised* learning strategy, using a mixture of labeled and unlabeled data [42]. Semi-supervised learning, such as the Expectation-Maximization (EM) method, uses unlabeled data for model improvement [42]. However, studies show that the usage of unlabeled data may not always help [35]. Thus, an empirical evaluation, suggested by recent studies [29], to investigate whether semi-supervised learning with unlabeled data actually helps is needed.

To address these challenges, in this paper, we evaluate two state-of-the-art deep learning methods: code2vec [3] and ASTNN [41], on the task of automatically detecting programming bugs in student code. We manually labeled three bugs in ∼1800 code submissions from 410 students in a Java programming course, where each bug occurred in 4-6 distinct problems. Our results show that, when using *all available training data*, the ASTNN model performs best at detecting all three bugs, outperforming code2vec and two classical baseline models (support vector machines and XGBoost).

Furthermore, we investigate whether a semi-supervised learning approach can improve the code2vec and ASTNN performance without requiring additional labeled data. More specifically, we investigated how the deep and baseline models performed with different amounts of labeled training data through a "cold start" analysis [32]. We found that all models benefited from more data. However, despite deep models' reputation as "data hungry," we found the top-performing model was generally a deep model, regardless of training data size. However, *which* model performed best depended on the data size, with code2vec outperforming ASTNN when less labeled data was available. We also found that semi-supervised learning generally improves both code2vec and ASTNN by using unlabeled data. This effect was most consistent for ASTNN, where semi-supervised learning consistently improved the model performance by 5% to 20% on all splits. For code2vec, we also found that it required very little data (5%) to achieve 80% of its peak F1 score.

The major contributions of this paper are addressing three research questions (RQs):

- RQ1: How well do state-of-the-art deep learning models for programming code perform in a student bug detection task?
- RQ2: How are deep learning models' performance impacted by the amount of available training data?
- RQ3: To what extent does semi-supervised learning improve the performance of the deep learning models?

2. APPROACHES

In this section, we introduce how we build code2vec and ASTNN for program classification; and how we applied the semi-supervised learning strategy on them to leverage unlabeled data.

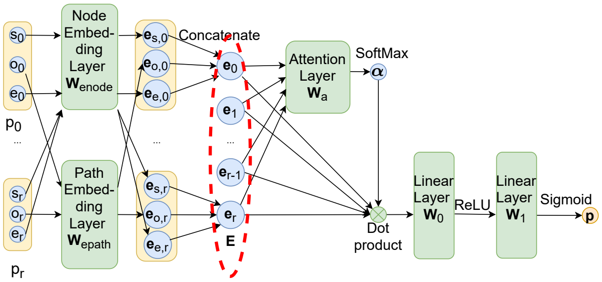

Figure 1: Code2vec model structure: model takes a set of paths as input, and through embedding layers, attention layer, then detect if the input code has bugs (1) or not (0).

2.1 Code2vec

One primary technical challenge in applying machine learning to program code lies in code representation. Code is often represented using an abstract syntax tree (AST) [7], while most learning algorithms expect a fixed-length vector. To solve this issue, sub-components of the ASTs are used as inputs for deep learning models. In the case of the code2vec model, it learns a code embedding through leaf-to-leaf paths, represented as strings. Strings of nodes and paths are mapped into numbers by tokenizers, where different strings are mapped into different numbers. These numbers are used as the input of a code2vec model, shown in Figure 1. Assume we have a code snippet that produces R paths $(p_0, ..., p_r)$ to be fed into the code2vec model. These numbers are embedded into e-dimensional vectors through node and path embedding layers ($\mathbf{W}_{enode}$ and $\mathbf{W}_{epath}$) respectively, and these node and path vectors are concatenated together into one vector for each of the paths $(\mathbf{e}_0, ..., \mathbf{e}_r)$. These vectors form a matrix $\mathbf{E}$, where $\mathbf{E} \in \mathbb{R}^{e \times R}$. Then these path vectors pass through a soft attention layer $\mathbf{W_a}$ [40], where they calculate the soft attention weight α for each of the paths: $\alpha = SoftMax(\mathbf{W_a}^\top \mathbf{E})$, $\mathbf{W_a} \in \mathbb{R}^{e \times 1}$, and thus α has scalar weights α_r for each of the paths, normalized by a SoftMax operator. Then the embedded path vectors $\mathbf{E}$ take the dot product of the calculated attention weights, showing which paths are more important in a code snippet. Then the weighted average vector passes through two fully-connected layers to make the bug classifications. In the training process, all the $\mathbf{W}$ weights are updated using Adam [23] optimization algorithm, while in the evaluation and validation processes, the weights in model are not changed.

2.2 ASTNN

Different from the path-based inputs for code2vec, ASTNN utilize the statement-level ASTs to learn a vector for the code. Specifically, we split the large AST of a code fragment by the granularity of the statement and extract the sequence of statement trees (ST-trees) with a pre-order traversal, and feed them as the raw input of ASTNN. Suppose that we have a set of ST-trees $(s_1, s_2, ..., s_T)$, our goal is to learn a vector representation z for the original code. The detailed architecture of ASTNN is shown in Figure 2.

Statement Encoder: Each ST-tree is composed of a root node and its child indices from a limited vocabulary of up to V symbols. For a ST-tree s_i, we first represent all nodes with

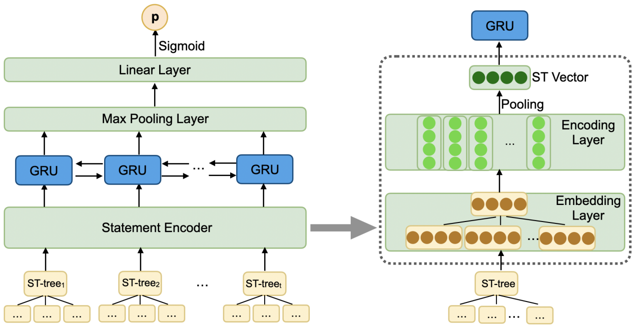

Figure 2: ASTNN model structure: model takes a set of statement trees as input, and through encoder layer, Bi-GRU layer, max-pooling layer, then to detect if the input code has bugs (1) or not (0).

the pre-trained embedding matrix $\mathbf{W}_{embed} \in \mathbb{R}^{V \times d}$ where V is the vocabulary size and d is the embedding dimension. Thus the initial vector of a node n can be obtained by:

$$\mathbf{v}_n = \mathbf{W}_{embed}^{\top} \mathbf{x}_n \tag{1}$$

where x_n is the one-hot encoding of node n. Next the ST-tree will go through a Recursive Neural Network [36] based encoder layer to update the vector for each node:

$$\mathbf{h}_n = \sigma(\mathbf{W}_{encode}^{\top} \mathbf{v}_n + \sum_{i \in child} \mathbf{h}_i + b_n) \tag{2}$$

where $\mathbf{W}_{encode} \in \mathbb{R}^{d \times k}$ is the encoding matrix and k is the encoding dimension. v_n is obtained from Equation 1 and b_n is the bias term. σ is the activation function and in this work we followed the original paper to set σ as identity function. After recursive optimization of the vectors of all nodes in the ST-tree, we sample the final representation e_i for s_i via a max-pooling layer.

Code Representation: Based on the sequences of ST-tree vectors, bidirectional GRU [5] is applied to track the naturalness of statements sequence $(e_1, e_2, ..., e_T)$, where T is the number of ST-trees in the AST:

$$h_i = [\overrightarrow{GRU}(e_i), \overleftarrow{GRU}(e_i)], i \in [1, L] \tag{3}$$

The statement representation $h_i \in \mathbb{R}^{L \times 2m}$, where m is the embedding dimension of GRU. Finally, similar to Statement Encoder, a max-pooling layer is used to sample the most important features on each of the embedding dimensions. Thus we get $z \in \mathbb{R}^{2m}$, which is treated as the final vector representation of the original code fragment. Finally $\mathbf{z}$ vectors pass through a linear layer to make the final classification of the bugs.

2.3 Semi-supervised Learning Strategy

While we explore the potential of machine learning models using insufficient labeled data as training inputs, unlabeled data can also serve as an important resource for the models to learn the structure of code. We applied a semi-supervised learning strategy to utilize these unlabeled data to help the model update. Specifically, in our experiments, we used Expectation-Maximization (EM) method [42] as an exploratory attempt. EM method is iterative, and it contains two steps for every iteration: 1) In expectation steps, the model infers on the unlabeled dataset, getting a probability score, which will be served as the pseudo-label in the next step, for each of the unlabeled code snippets; 2) In maximization steps, the model is retrained using all the labeled training dataset and the unlabeled set with the pseudo-labels from expectation step. After retraining, the model is used for the next round of expectation step. In our case, deep learning models are designed to output probability scores, but SVM and XGBoost models make classifications without clear scores or probabilities. We implemented the regression versions of the models, assuming they would output a continuous probability as the regression result. We then used 0.5 as the probability threshold to binarize the output, serving classification results. Every model uses a unified 10 iterations of EM steps, assuming the models are able to converge after a certain number of iterations and retraining.

3. EXPERIMENT SETTINGS

3.1 Dataset and Bug Labeling

We performed bug classification on a publicly available dataset, collected from an entry-level Java programming class in Spring 2019[1]. It was collected from the CodeWorkout [15] platform and stored in ProgSnap2 [31] format. Since Java compiler can already detect bugs from code that failed to compile (due to syntax errors), and this code cannot be converted into an AST, we excluded uncompilable code from our analysis. We also did not use code that passed all test cases, as this code is correct and therefore is very unlikely to have bugs. There are 410 students, who attempted in total 50 problems from 5 assignments. Typical solutions for these assignments range from 10 to 20 lines of code. In order to determine the common set of bugs across different problems, two authors examined student code from six distinct programming problems from the first assignment and identified common bugs that arose. They then selected 3 prevalent ones after calculating the coverage of bugs from each problems, and identified in prior CS education literature [17, 20]. This included 2 logical bugs and 1 syntax bug: `comparison-off-by-one` (logical), `assign-in-conditional` (syntax), and `and-vs-or` (logical), defined below:

`comparison-off-by-one`: This bug occurs if, in a conditional expression (e.g. in an `if` or `while`), the student's code uses a greater/less-than comparison operator (`<=`, `>=`, `<`, `>`) incorrectly, and this error can be resolved by adding or removing the '`=`', (e.g. `<` becomes `<=`). The direction of comparison (i.e. `<=` vs `>=`) should already be correct. This often indicates an "off by one" error, and it is contextual, dependent on the number of literals being compared. If there are multiple bugs, including this bug, we still count it.

`assign-in-conditional`: This bug occurs if, in a conditional expression, a student uses the `=` *assignment* operator in their code when trying to compare a variable with another value, rather than the correct `==` *comparison* operator. This is a syntax-based bug, but it is not detected by the compiler, since the assignment is logically a valid operation.

`and-vs-or`: This bug occurs if a student uses the logical

[1] https://pslcdatashop.web.cmu.edu/Project?id=585

Table 1: Detection performance for four classifiers on three bugs.

	Method	Accuracy	AUC	Precision	Recall	F1 Score (Std)
comparison-off-by-one	**SVM**	0.753	0.658	0.731	0.100	0.173 (0.045)
	XGBoost	0.505	0.541	0.384	0.547	0.334 (0.088)
	Code2Vec	0.736	0.746	0.500	0.556	0.522 (0.058)
	ASTNN	0.785	0.704	0.606	0.533	**0.560 (0.090)**
assign-in-conditional	**SVM**	0.943	0.959	0.918	0.627	0.733 (0.099)
	XGBoost	0.847	0.877	0.494	0.726	0.563 (0.112)
	Code2Vec	0.917	0.907	0.725	0.688	0.672 (0.119)
	ASTNN	0.970	0.901	0.961	0.807	**0.868 (0.094)**
and-vs-or	**SVM**	0.722	0.674	0.534	0.173	0.256 (0.078)
	XGBoost	0.503	0.669	0.350	0.784	0.470 (0.045)
	Code2Vec	0.758	0.821	0.570	0.663	0.609 (0.078)
	ASTNN	0.880	0.837	0.820	0.739	**0.773 (0.064)**

operator `and` instead of `or` in their code, or vice-versa, such that the opposite operator would produce correct code. This is also a logical bug that requires contextual information but is easier to detect than `comparison-off-by-one`. It requires the literals, but does not depend much on problem requirements.

Two authors started by labeling 20% of the data, following the same set of initial bug definitions. The labeling process was iterative: the two authors first labeled 20% of the data independently and then calculated Cohen's Kappa scores κ. If on any of the three bugs, the two authors did not achieve a score higher than the 0.8 [24], then the authors discussed and resolved the disagreements, refined the definitions, and continued to another round of independently labeling 10% of the data. This process continued until the authors reached high agreement ($\kappa > 0.8$) on all three categories of bugs, which occurred after labeling 40% of the data. The first two rounds of labeling did not achieve a high κ score, both due to the low scores on the `comparison-off-by-one` bug, suggest that this bug may be more difficult to consistently detect for humans. On the third round of labeling, the two authors achieved 0.81, 0.97 and 0.84 κ scores on the three bugs. Then the authors divided the rest 60% of data by 35% for each person to label, overlapping on 10% of the data for verification. These 10% of data achieved 0.78, 0.98 and 0.95 κ scores, indicating moderate to near-perfect agreement [27]. The finalized labeled dataset has a biased distribution, as only 30% of the submissions have `comparison-off-by-one` bug, 28% of the submissions have `and-vs-or` bug, and 13% have `assign-in-conditional` bug. In total, we spent around 20 hours and labeled 1867 code snippets from 296 students.

3.2 Splits in Experiments

Since our dataset included multiple attempts from a given student, we split our data into training and testing sets by student. This ensured that a given student's code showed up in either the training *or* testing set, but not both. In our experiment, we have 20% of the data as the test set, and the rest 80% are used for model generation. To check the performance of models with limited labeled data, we further split the 80% of data into labeled data and unlabeled data. We use only labeled data for supervised learning, and use both labeled and unlabeled data for semi-supervised setting. All these splits were stratified according to the class label and number of submissions, ensuring that a similar proportion of buggy/non-buggy programs were in each split. This is necessary, since splitting by students can create very biased distributions, especially when we only have small labeled training sets. The stratification uses thresholds for 1) the ratio of bugs and 2) averaged submission numbers for students in respective bug groups. We argue that in practice, we should be able to select a similarly representative sample by manually checking several submissions to see if the distribution is fundamentally different. To ensure we evaluate our model performance with fair comparisons, we created 10 different splits, generated randomly. All models use the same training/testing splits, and average performance metrics are reported as the results. For semi-supervised setting, we varied the size of labeled/unlabeled data to evaluate the performance of models. In order to perform fair comparisons, all semi-supervised models have the same labeled/unlabeled splits. Also, all models are tested on the same test sets, regardless of the model, the amount of training data, or supervised vs. semi-supervised. These settings ensured fair comparisons across different models.

3.3 Model Settings

SVM and XGBoost Parameters: We performed grid search on hyperparameters for SVM and XGBoost models using cross-validation on the training sets. In the SVM setting, we searched linear and Radial Basis Function (RBF) kernels, with C parameters in a range of (0.1 - 1), stepping by 0.1. In the XGBoost setting, we searched through situations that sub-sample portions from 0.1 to 1, stepping by 0.1, using 5 to 100 estimators in the model. To prepare numerical input, we used TF-IDF feature extraction on the code submissions for both models.

Code2vec and ASTNN Parameters: Since deep learning models are more time- and resource-consuming, and our cold start experiments required many repeated runs (~ 100 runs), we did not perform automatic grid search; rather, we used default settings of the hyper-parameters and did manual changes. In code2vec, after observing the training and validation loss, we set the maximum training epochs as 200, with the *patience* of early stopping set to 100, and set the learning rate to 0.0002. Linear layer and embedding dimensions were kept at the default value of 100. To ensure the highest efficiency of the model, we set the batch size as the full batch. These parameters are tuned with different numbers, but little change in validation accuracy is observed. We also manually padded the number of paths to 100 over all code submissions. In ASTNN, we padded the statement sequences to the maximum length to accommodate the longest sequence before feeding to Bi-GRU. During training, we used 32 as batch size, 0.001 as learning rate, and keep the max training epoch as 50. The encoding dimension for the statement encoder was 128, and hidden neurons for Bi-GRU were 100. The weights were learned during training using the Adam optimizer for code2vec and ASTNN models.

4. RESULTS

4.1 Bug Detection Model Performance

In this subsection we address RQ1: *How well do state-of-the-art deep learning models for programming codes perform in a student bug detection task?* Table 1 shows the results of the classifiers in the task of detecting bugs across problems. We use accuracy, Area-under-curve (AUC), Precision (P), Recall (R), and F1 score as the evaluation metrics for the

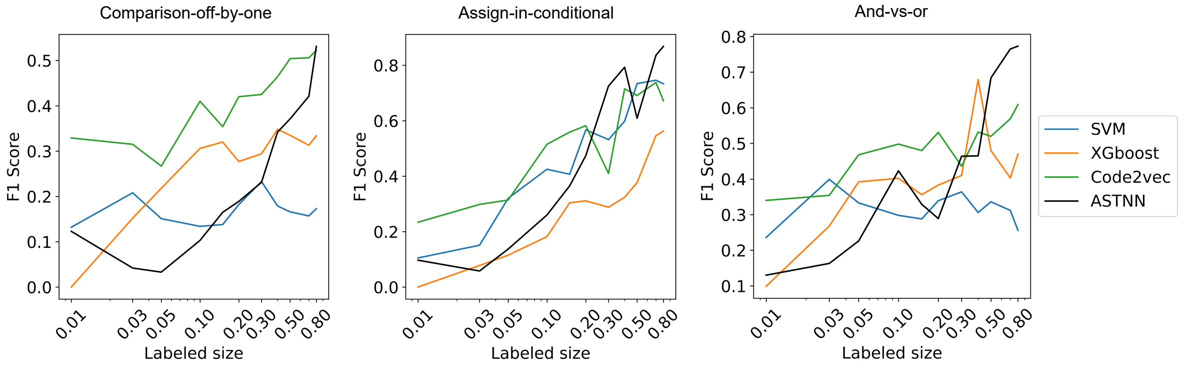

Figure 3: F1 score of models using supervised strategy with different portions of labeled data in detecting bugs.

detection. Across the three bugs for detection, we observe that the top-performing model (ASTNN) achieved 0.560, 0.868, 0.773 F1 scores on detecting `comparison-off-by-one`, `assign-in-conditional`, and `and-vs-or` respectively. The F1 scores indicate that the models achieved a higher performance on detecting `assign-in-conditional` compared to `comparison-off-by-one` and `and-vs-or`. In all bug detection tasks, ASTNN achieved the best F1 score on all bugs. On the two logical bugs, ASTNN achieved at least 0.226 higher F1 score than SVM and XGBoost, while Code2vec also achieved at least 0.139 higher, showing deep learning models are more preferable for detecting the two logic bugs across the six problems. On the detection of the `assign-in-conditional` bug, ASTNN achieved a high F1 score of 0.868, while a simple SVM model is able to achieve a 0.733 F1 score, which is not much lower than the ASTNN model. However, the recall of SVM is low (0.627), which indicates a limited capability of detecting the bugs out of submissions with bugs. Code2vec model did not achieve better F1 or AUC scores than SVM or ASTNN model in this case, showing that in the detection of syntax issues, paths features might be overly complicated. SVM might have a good performance when the real rule to learn is just "If it has `=` instead of `==` in the code, it has the bug," since there is little contextual information to learn. Generally speaking, when using all 80% labeled dataset (∼ 1493 programs on average), deep learning models have a better performance than traditional machine learning models in detecting logical bugs, showing the advantage of leveraging structural information in the feature extraction step.

4.2 Bug Detection with Limited Labels

We address RQ2 in this subsection: *How are deep learning models' performance impacted by the amount of available training data?* From Figure 3, we see the F1 scores of the four models in supervised strategy using a subset of labeled data. The x-axis is the log-scaled labeled data size, and the y-axis is the F1 score that models achieved across the 10 different splits. The lowest portion of labeled data we use is 1%, which contains around 15 students, while the highest portion is 80%. The general trend of the supervised models shows that when more data is used, better F1 scores can be achieved by models, especially ASTNN. We also observe some interruptions in the increment of the performance as more data is available, meaning that it is not guaranteed that more data generate better models. For other baseline models, such a data-performance relationship is weaker, but still more data can generally produce better models.

While the models expect better performance given more data, we would like to note that among all supervised models, code2vec achieved better results than other models using a small subset of labeled data, showing a property of warm starting. With 10 percent of labeled data, code2vec has at least 7.5% higher F1 scores than any other models on all the three detected bugs. When more data is used, ASTNN outperforms other models, showing that there is generally at least one deep learning model more preferable than baseline models. When comparing code2vec with ASTNN, we find that deep learning models are not always "data-hungry": although both models are are sensitive to data size, code2vec starts higher than baseline models in classifying all three bugs. To achieve a good detection result, using 30%-40% (560-747) less labeled data would create models achieving 80% of the F1 score.

With these results we are able to conclude the answer for RQ2: For code2vec and ASTNN, more data would produce models with better performance. However, the relation is not linear: ASTNN is more "data-hungry" than code2vec, but these deep learning models do not require lots of data points to perform better than baselines.

4.3 Application of Semi-supervised learning

This subsection addresses RQ3: *To what extent does semi-supervised learning improve the performance of the deep learning models?* Figure 4 shows the semi-supervised learning results for all four models and the comparisons to supervised ASTNN and code2vec models. The labeled training data for each split is exactly the same as ones used in supervised settings. While the results give a mixed signal about whether semi-supervised learning is beneficial for all models, we have two observations. 1) semi-supervised learning enhanced the learning of *deep models*, especially ASTNN in all three bugs. Comparing the black lines, we found that solid lines are always higher than dashed ones. It suggests ASTNN, as a more "data-hungry model", is favored by the semi-supervised strategy more than in other models. Typically, an ASTNN model trained with a semi-supervised learning strategy achieves 0.05 to 0.2 higher F1 scores than

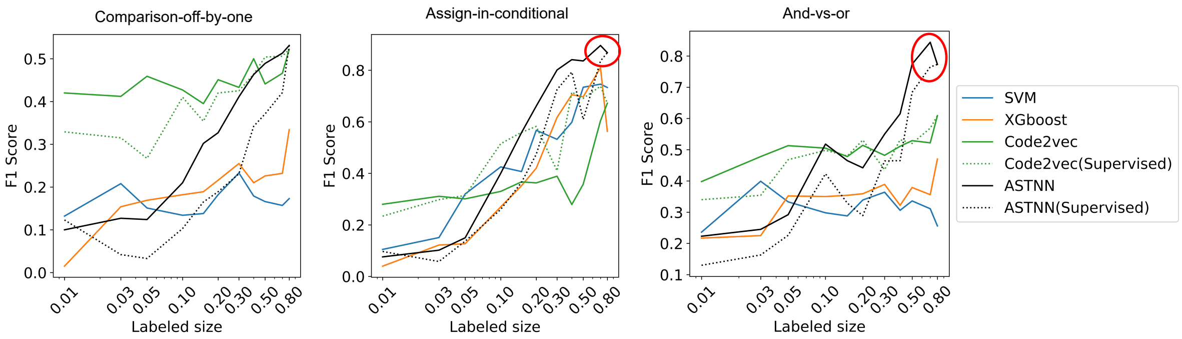

Figure 4: F1 score of models using semi-supervised strategy with different portions of labeled data in detecting bugs. Red circles noted places when semi-supervised strategy outperformed supervised training with full data.

those trained in supervised learning strategy, using the same training dataset. In code2vec, which is also a deep learning model, semi-supervised learning does not always help. It helps code2vec achieve better F1 scores when using a lower portion of data, but when given more data, a supervised learning strategy provides better performance. Semi-supervised learning does not help much for the other two classical models, compared with deep models. 2) In semi-supervised learning scenario, ASTNN achieved a better performance when using 70% labeled data than using all 80% as training in detecting two bugs, `assign-in-conditional` and `and-vs-or`, by 2.8% and 7.1% respectively (red-circled in Figure 4). While this may reflect the fluctuation of data performance, we did run the models 10 times. This suggests that the model may be harnessing the semi-supervised learning strategy to infer labels for unlabeled sets, and achieve more consistent labels than the authors, or some outliers present in the unlabeled set. We assume that the model then learned on these automatically inferred labels and achieved better results than learning from all expert labeled data.

Our conclusion for RQ3 is that semi-supervised learning often improves performance, especially when little training data is available. It enables the models to achieve an expected performance with less labeled data than the supervised scenario. Specifically, semi-supervised learning helped all cases in the learning of ASTNN models, and helped code2vec overall as well, especially when data size is low.

5. DISCUSSION CONCLUSION

Our results suggest three primary conclusions: **1) The two deep learning models generally outperformed baselines, and ASTNN had the best performance.** Our results from Subsection 4.1 show that deep learning models can detect simpler bugs, but still have a limited effectiveness on more complicated bugs (detailed in Subsection 3.1). The complexity of the `comparison-off-by-one` bug may be due to the difficulty of the labeling process, or its dependency on the problem context. **2) Deep learning models may still be successful when labeled data is limited**. From the results in Subsection 4.2, we learn that even if training with small data size such as <100 data points in complicated programming data, the code2vec model is still able to outperform baseline models. **3) Semi-supervised learning has the potential to help deep learning models perform better.** Semi-supervised learning helped code2vec to achieve a higher performance, but only when a small number of data points are labeled. One may assume the difference between the two deep learning models come from the structures, but it may also come from the feature extraction process. Code2vec uses paths based features but ASTNN uses node based features, and recursively processed by neural networks.

Our results can have other potential applications in educational program analysis tasks as well. For example, as features are automatically extracted from student code during code2vec or ASTNN training, these features can be used to help instructor discover *new bugs*, as suggested by [33], which can help shape instruction. If more features such as problem requirements and test case inputs are available, we can apply these features to the model introduced by [30] to propagate instructor feedback to all students who would benefit form it.

This work also has a couple of caveats or limitations as of the current stage. 1) We only performed extensive experiments on three bugs and used them to generalize to conclusions. This is because the dataset labeling is time consuming, requiring the authors to label ~ 1800 data points. The conclusions here may not generalize to other bugs or code classification tasks. 2) Similarly, these bugs also come from one programming assignment near the beginning of the course, focused on if conditions, and thus may be biased to this specific type of problem. 3) In the splitting process, we performed stratified sampling, requiring that test, labeled, and unlabeled data be a similar distribution of class labels and the number of attempts. 4) Since we only compared our models with two classical model baselines, there may be other better models existing for better performance. We used our best effort to select representative models that achieve state of the art performance, but there might be better models available for the task as well. This work's primary goal is to lay the foundation for using deep models in this task by exploring if the "data-hungry" property also applies here, and potential applications of semi-supervised learning. It serves as a step towards future model designs specific for automatic student bug detection, and provides guideline for situations when labeled data is limited.

Acknowledgements: This research was supported by the NSF Grants: #1623470, #1726550, #1651909 and #2013502.

6. REFERENCES

[1] C. C. Aggarwal et al. Neural networks and deep learning. *Springer*, 10:978–3, 2018.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A general path-based representation for predicting program properties. *PLDI'18*, 2018.

[3] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *POPL'19*, 2019.

[4] P. Baffes and R. Mooney. Refinement-based student modeling and automated bug library construction. *Journal of Artificial Intelligence in Education*, 7(1):75–116, 1996.

[5] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] A. Bajwa, A. Bell, E. Hemberg, and U.-M. O'Reilly. Analyzing student code trajectories in an introductory programming mooc. In *2019 IEEE Learning With MOOCS (LWMOOCS)*, pages 53–58. IEEE, 2019.

[7] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[8] J. Bonar and E. Soloway. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction*, 1(2):133–161, 1985.

[9] J. S. Brown and K. VanLehn. Repair theory: A generative theory of bugs in procedural skills. *Cognitive science*, 4(4):379–426, 1980.

[10] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings. 26th International Conference on Software Engineering*, pages 480–490. IEEE, 2004.

[11] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, et al. Xgboost: extreme gradient boosting. *R package version 0.4-2*, 1(4), 2015.

[12] J. Choi, H. Kim, C. Choi, and P. Kim. Efficient malicious code detection using n-gram analysis and svm. In *2011 14th International Conference on Network-Based Information Systems*, pages 618–621. IEEE, 2011.

[13] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[14] A. Dutt, M. A. Ismail, and T. Herawan. A systematic review on educational data mining. *Ieee Access*, 5:15991–16005, 2017.

[15] S. H. Edwards and K. P. Murali. Codeworkout: short programming exercises with built-in data collection. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 188–193, 2017.

[16] S. H. Edwards and M. A. Perez-Quinones. Web-cat: automatically grading programming assignments. In *ITiCSE'08*, pages 328–328, 2008.

[17] A. Ettles, A. Luxton-Reilly, and P. Denny. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 83–89, 2018.

[18] A. Gupta, S. Sharma, S. Goyal, and M. Rashid. Novel xgboost tuned machine learning model for software bug prediction. In *2020 International Conference on Intelligent Engineering and Management (ICIEM)*, pages 376–380. IEEE, 2020.

[19] R. Gupta, A. Kanade, and S. Shevade. Neural attribution for semantic bug-localization in student programs. *Network*, 1(P2):P2, 2019.

[20] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[21] A. Kaur, S. Jain, and S. Goel. A support vector machine based approach for code smell detection. In *2017 International Conference on Machine Learning and Data Science (MLDS)*, pages 9–14. IEEE, 2017.

[22] H. Keuning, J. Jeuring, and B. Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 41–46, 2016.

[23] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[24] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[25] M. H. Liu. Blending a class video blog to optimize student learning outcomes in higher education. *The Internet and Higher Education*, 30:44 – 53, 2016.

[26] Y. Mao, S. Marwan, T. W. Price, T. Barnes, and M. Chi. What time is it? student modeling needs to know. In *In proceedings of the 13th International Conference on Educational Data Mining*, 2020.

[27] M. L. McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012.

[28] 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.

[29] A. Oliver, A. Odena, C. A. Raffel, E. D. Cubuk, and I. Goodfellow. Realistic evaluation of deep semi-supervised learning algorithms. *Advances in Neural Information Processing Systems*, 31:3235–3246, 2018.

[30] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *International Conference on Machine Learning*, pages 1093–1102, 2015.

[31] T. W. Price, D. Hovemeyer, K. Rivers, G. Gao, A. C. Bart, A. M. Kazerouni, B. A. Becker, A. Petersen, L. Gusukuma, S. H. Edwards, et al. Progsnap2: A flexible format for programming process data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 356–362, 2020.

[32] T. W. Price, R. Zhi, Y. Dong, N. Lytle, and T. Barnes. The impact of data quantity and source on the quality of data-driven hints for programming. In *International conference on artificial intelligence in education*, pages 476–490. Springer, 2018.

[33] Y. Shi, K. Shah, W. Wang, S. Marwan, P. Penmetsa, and T. Price. Toward semi-automatic misconception discovery using code embeddings. In *The 11th International Conference on Learning Analytics Knowledge (LAK 21)*, 2021.

[34] V. J. Shute. Focus on Formative Feedback. *Review of Educational Research*, 78(1):153–189, 2008.

[35] A. Singh, R. D. Nowak, and X. Zhu. Unlabeled data: Now it helps, now it doesn't. In *NIPS*, volume 21, pages 1513–1520, 2008.

[36] R. Socher, C. C.-Y. Lin, A. Y. Ng, and C. D. Manning. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 2011.

[37] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM Sigcse Bulletin*, 38(3):13–17, 2006.

[38] K. VanLehn. *Mind bugs: The origins of procedural misconceptions*. MIT press, 1990.

[39] S. Wiedenbeck, V. Fix, and J. Scholtz. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 39(5):793–812, 1993.

[40] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2048–2057, Lille, France, 07–09 Jul 2015. PMLR.

[41] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[42] X. Zhu and A. B. Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130, 2009.