Predicting Bug Fix Time in Students' Programming with Deep Language Models

Stav Tsabari
Ben-Gurion University
stavts@bgu.ac.il

Avi Segal
Ben-Gurion University
avise@post.bgu.ac.il

Kobi Gal
Ben-Gurion University
University of Edinburgh
kobig@bgu.ac.il

ABSTRACT

Automatically identifying struggling students learning to program can assist teachers in providing timely and focused help. This work presents a new deep-learning language model for predicting "bug-fix-time", the expected duration between when a software bug occurs and the time it will be fixed by the student. Such information can guide teachers' attention to students most in need. The input to the model includes snapshots of the student's evolving software code and additional meta-features. The model combines a transformer-based neural architecture for embedding students' code in programming language space with a time-aware LSTM for representing the evolving code snapshots. We evaluate our approach with data obtained from two Java development environments created for beginner programmers. We focused on common programming errors which differ in their difficulty and whether they can be uniquely identified during compilation. Our deep language model was able to outperform several baseline models that use an alternative embedding method or do not consider how the programmer's code changes over time. Our results demonstrate the added value of utilizing multiple code snapshots to predict bug-fix-time using deep language models for programming.

Keywords

computer programming, predict bug fix time, deep learning language models

1. INTRODUCTION

Programming courses have become an essential component of many STEM degrees and are attracting students from diverse backgrounds. Many beginners struggle with learning fundamental principles of programming [20]. Additionally, previous studies suggest that compiler messages have an imperfect mapping to errors which can confuse the students [24]. Providing students with personalized support can significantly aid their learning. The time spent by programmers to fix bugs is known to be a proxy for the difficulty they encounter and can be used as an indicator for finding struggling students [14, 3]. Thus, predicting the bug-fix-time of errors for students can help teachers identify those students requiring additional attention and support. Such prediction can also support hint generation systems for better inferring when a hint is needed [11, 23].

Past work has estimated the bug-fix-time for different errors based on bug error reports. These are reports created by the quality assurance team in organizations to describe and document the bugs found in computer programs [15, 33, 18]. These studies ignored the personal variations between programmers, predicting a single value per a specific bug.

We address this gap by providing a personalized approach for predicting bug-fix-time for programming errors. Our underlying assumption is that if the student's fix time for a bug is longer than a threshold, it may indicate a struggling student requiring assistance and guidance. Specifically, our method predicts if the error fix time will be "short" or "long", with the median used as the cutoff value. The median is chosen as threshold to focus on the lower half of students that may benefit from some level of assistance. This is a standard approach in other works studying bug-fix-time [5, 15].

Our approach is personalized per student and per bug type and uses snapshots of the evolving student's code. Errors vary in whether the compiler can identify the error, and whether the compiler's error message is unique to the specific error type. The proposed method is based on CodeBert[13], a state-of-the-art transformer-based neural architecture for embedding students' programming code, and combines an LSTM-based architecture which is used to capture multiple time dependant code snapshots.

We compared our approach to three baselines for predicting the fixed time of the different errors in two datasets (1) A method that is based on the Halstead Metrics [16]. This approach computes features based on operators and operands in the code. (2) A code embedding-based approach using Code2Vec [2], which is a common framework for learning representations of natural language and code, and has been used previously in an educational context. This approach considered the student's code which produced the bug as well as the prior code submissions of the same program. (3) A language model-based approach using CodeBert which considered only the student's code that produced the bug (4) Our approach: A language model-based approach using

S. Tsabari, A. Segal, and K. Gal. Predicting bug fix time in students' programming with deep language models. In M. Feng, T. Käser, and P. Talukdar, editors, *Proceedings of the 16th International Conference on Educational Data Mining*, pages 396–405, Bengaluru, India, July 2023. International Educational Data Mining Society.

© 2023 Copyright is held by the author(s). This work is distributed under the Creative Commons Attribution NonCommercial NoDerivatives 4.0 International (CC BY-NC-ND 4.0) license.
https://doi.org/10.5281/zenodo.8115733

CodeBert which considered the student's code which produced the bug as well as the previous code snapshots saved by the system while the student was evolving their code.

We evaluated our approach on code and compilation instances obtained from thousands of students' code submissions, sampled from two different programming environments. The first environment was the BlueJ Java development environment [21], a programming environment designed for beginner programmers and used in a large number of educational institutions. We obtained 241,418 code submission instances containing errors that were generated by students when learning to program. The prediction task focused on 4 common types of novice errors that differ in complexity.

The second environment, called CodeWorkout, was collected from an introductory programming course in the Spring and Fall of 2019 semesters at a public university in the U.S. [12]. We obtained 80,013 code submission instances that contained 75 different compilation errors, each error has a different cause, such as: unknown variable, missing operands etc. The CodeWorkout dataset contained simpler computing problems, with typically shorter submitted programming solutions.

We considered two different settings for the BlueJ dataset, one in which a different model was built for each error type and a setting where one model was built for all error types. The CodeWorkeout dataset was tested with one model per all error types due to data size limitations.

For both environments, our proposed approach was able to outperform the baseline approaches in terms of accuracy, recall, and F1 measures when predicting bug-fix-time. These results demonstrate the efficiency of using transformer-based language models developed for programming languages for solving the bug-fix-time prediction task and the value of adding students' code history for such tasks. Our approach has implications for software development education, in that it can potentially be used by instructors to identify struggling students requiring further support.

2. RELATED WORK

Our work relates to several research areas: (1) Programming errors performed by novice programmers (2) Predicting bug-fix-time for programming errors, and(3) Recent deep Natural Language Processing language models for programming language representations. We elaborate on each one in turn.

2.1 Student Programming Errors

Hristova et al. [19] collected a list of common students' Java programming errors based on reporting of teaching assistants. Most of the errors identified were detected and reported by the Java compiler. Nonetheless, McCall et al. [24] investigated logical errors in students' code and suggested that compiler messages alone have an imperfect mapping to student logical errors. They demonstrated that the same logical error can produce different compiler error messages and different logical errors can produce the same compiler error message. Bayman et al. [8] examined errors related to individual program statements and found that many learners possess a wide range of misconceptions about individual statements or constructs of even very simple statements, which lead to programming errors.

Brown et al. [4] analyzed 18 students' errors using the BlueJ dataset and focused on two error types, those identified by the compiler, and those that require a customized source code analyzer that searches the source code for programming mistakes. Errors relating to the latter type are not uniquely identifiable by the compiler and may be more complex to fix. We are inspired by this study and test the potential of machine learning-based approaches to predict the bug-fix-time of student errors for both compiler errors as well as errors identified by static source code analysis.

2.2 Predicting Bug Fix Time of Programming Errors

Various approaches have been used in past research to predict the time required for fixing bugs. Zhang et al. [32] investigated the connection between bug reports and other features and the bug fixing time. Bug reports are the reports created by the quality assurance and testing team in an organization to describe and document the bugs found in a computer program and include attributes such as problem description and priority. Zhang et al. [33] predicted the number of bugs to be fixed and estimated the time required to fix a certain bug using bug report attributes only. They estimated the time to fix a bug as "slow" or "quick" based on several thresholds. Other studies have focused on predicting bug fixing time using different classifications than "slow" or "fast". Panjer et al. [25] employed multi-classification using various classification models to classify the time to fix bugs into seven-time buckets, using only the bug report.

Some studies have focused on predicting the exact time to fix the bug. Weiss et al. [29] used text similarity to predict the bug-fixing time. Given a new bug report, they used text similarity to search for similar, earlier reports and use their average time as the prediction time. Recently, some deep network-based approaches were proposed for the bug-fix-time prediction problem. Ardimento et al. [5] used BERT, a pre-trained deep bidirectional Transformer model, to predict bug fixing time as fast or slow from bug reports. This approach has shown the best performance so far.

Our research differentiates from these past efforts in three main manners: (1) First, all past work performed non personalized bug-fix-time prediction. I.e., the prediction was performed per error type and not per user. In contrast, we focus on predicting bug-fix-time per user for each error type. (2) Second, past studies used errors of experienced programmers and were trained on code repositories such as GitHub and the like. In this research, we focus on errors generated by novice student programmers and use appropriate datasets for this task. (3) Third, past studies did not directly take into account the programmer's source code nor did they use previous code snapshots which capture the programmer's evolving code prior to the error generation. Specifically, these works used only attributes from bug reports and did not directly consider the code in which the bug was found. In this paper, we hypothesize that the source code itself as well as past code snapshots of the programmer's evolving work hold strong signals for predicting the bug-fix-time for errors generated by the programmer.

2.3 Language Models for Programming Language Representations

One technique for the embedding of software program methods is Code2Vec, a neural model for representing snippets of code as continuously distributed vectors [2]. Code2Vec was developed for the task of method name prediction and uses paths in the program's abstract syntax tree (AST) for its embeddings [7]. We choose to use Code2Vec as a baseline since it outperformed other models in past works and has been used previously in educational contexts [26, 6].

Recently, deep language models have been developed for code representations. One such model which demonstrated state-of-the-art results is CodeBert [13]. CodeBert is a transformer based large scale language model for both natural and programming languages. The model is trained with a dataset that includes 6.4M unimodal source codes in different programming languages including Java, Python, Go, JavaScript, PHP, and Ruby. CodeBert learns general-purpose representations and can be fine-tuned to support downstream natural language and programming language applications such as source code classification tasks. We used CodeBert for code representation in our proposed approach.

3. METHODOLOGY

In this study, we evaluated the usage of a large scale deep learning language model combined with software code snapshots for the prediction of bug fix time. Specifically, our research questions were as follows: (**RQ1**) Do models that embed students' code do better than those relying on Halstead metrics features when predicting bug fix time? (**RQ2**) Can deep language models built for software code representation improve such a prediction task? (**RQ3**) Does using preceding snapshots of the students' code further improves the prediction task?

3.1 Datasets

To increase the generality of the developed approach, two datasets were used in this study, both from development environments created for the Java programming language. The first dataset was obtained from the BlueJ environment which is a general-purpose Java programming environment designed for beginner programmers. The second dataset was collected on the CodeWorkout environment which contains assignments submitted by Java students during two semesters. We note that the obtained data from both programming environments did not include any personal or demographic information about users. In both datasets, we leave out errors that were not solved altogether by the student (i.e., the bug fix time is unknown).

3.1.1 The BlueJ Dataset

BlueJ is an integrated Java programming environment designed for beginners and used in a large number of institutions around the world [21]. The environment has been used for a variety of assignments designed to support and exploit pedagogical theories of programming. The BlueJ platform includes an advanced capability of recording the student's programs (as they are being developed) in a dedicated research environment called Blackbox [9]. In Blackbox, each instance includes a timestamp of a compilation event by a programmer, together with the code that was submitted for that compilation, a student ID, a session ID, and a list of error messages reported by the compiler (if any).

Table 1: Selected Errors in the BlueJ Dataset

Bug Type	***Median BFT(sec.)***	***Average BFT(sec.)***	***STD BFT(sec.)***	***Num. instances***
I	51	164	261	80,000
O	35	136	239	80,000
A	52	193	298	60,254
B	60	228	398	21,164

Similarly to Brown et al.[4], we used a dataset representing one year of activity in the BlueJ environment, from September 1st, 2013 to August 31st, 2014. In this set, we focus on the four errors that were identified by Brown et al.[4] among the most common errors for novice programmers:

Error I: Invoking method with a wrong argument type; the compiler can uniquely detect this error.
Error O: Non-void method without a return statement; the compiler can uniquely detect this error.
Error A: Confusing operator (=) with (==); the compiler detects error, but does not output a unique error statement.
Error B: Using the operator (==) instead of (.equals); the compiler cannot detect the error.

In this research, errors I and O were identified directly from the output messages issued by the compiler. Errors A and B were identified by a static analyzer built using XML representation [10] of projects' code. In total, the dataset contains $17,682,006$ instances. From this dataset, we sample $241,418$ instances which include the four mentioned error types. Each instance in the dataset is a code submission.

Table 1 presents the number of instances and the bug-fix-time (BFT) median, average, and STD values for each selected error in the BlueJ platform. As shown by the table, the errors vary in average difficulty in terms of the average fix time. An example of the distribution of bug-fix-time for error I in BlueJ can be seen in Figure 1. As seen in the figure, the distribution is right-skewed, with some students exhibiting very long fix times for this error. A similar trend was also apparent for the other bug types in the platform.

3.1.2 The CodeWorkout Environment

The CodeWorkout environment [12] is an online system for people learning Java programming for the first time. This open-source site contains 837 coding problems spanning topics such as sorting, searching, and counting. The environment includes tests for each problem, which verifies the correctness of each student's submission. Student submissions are graded automatically using these tests and feedback is returned including error messages.

The dataset includes student assignment submissions from an introductory course of Computer Science course ("CS1") administered in the Spring and Fall 2019 semesters at a public university in the U.S. During this course, 50 different coding problems from CodeWorkout were given to students,

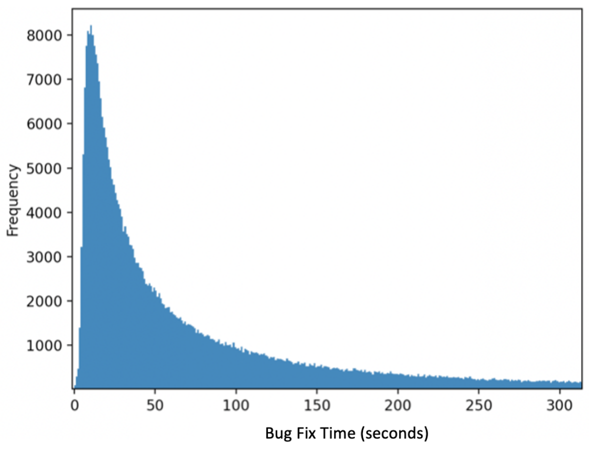


Figure 1: Distribution of bug-fix-time for Error I in BlueJ

and their code submissions were collected. The coding problems used in this dataset are easy and designed for beginner programmers. Each problem requires 10-26 lines of code and includes basic operations such as for loops, and if / else statements. Each submission in the CodeWorkout dataset includes the submitted code by the user, the user ID, the assignment ID, and the list of reported compiler errors if any. The CodeWorkout dataset included only bugs that are identifiable by the compiler. Overall this dataset contains 75 error types from 819 students and a total of 80,013 instances. As with the BlueJ dataset, the bug-fix-time in the CodeWorkout datasets exhibits a long tail distribution.

3.2 Computing Bug-Fix-Time

The bug-fix-time (BFT) for a given error is defined as the length of time (in seconds) between the first compilation submission in which the error was reported, and the nearest compilation submission in which the error was resolved. For the BlueJ environment, The bug-fix time is adjusted to the periods when the user is logged into the system. For CodeWorkout, session login information is not available, so bug-fix-time considers only bug occurrence and bug resolution timestamps.

We note the high variance in bug fixing time as indicated by the STD value in table 1 (also occurs for CodeWorkout). This indicates that bug-fixing time is a personal phenomenon, which may depend on specific students' characteristics. We hypothesize that such characteristics are expressed in the way that students write and evolve their program code. Thus, we define the bug-fix-time prediction problem, as the problem of predicting the time to fix a bug for a **specific student** given the occurrence of a **specific bug** in their latest compilation and considering their past code submissions. Following past approaches, we use the median value of bug fixing time per each bug type as the threshold between "slow" and "fast" fixing times [5].

3.3 Predicting Bug Fix Time

We now describe the deep learning model for the prediction task at hand. Our architecture includes three layers: (1) An embedding layer using pre-trained deep language models for representing programming languages, (2) A time-aware layer using Long Short Term Memory, and (3) A classification layer. The architecture can be seen in Figure 2.

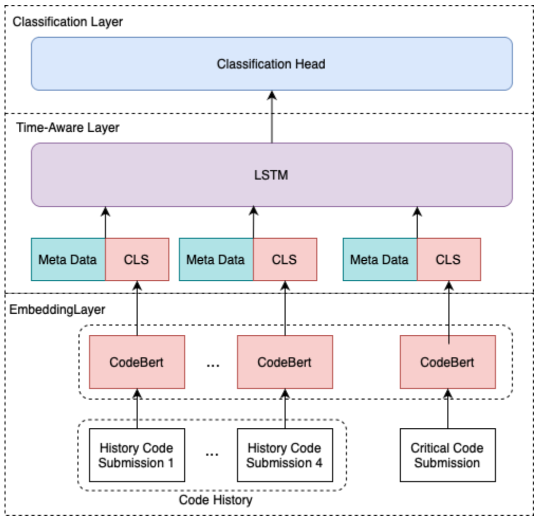


Figure 2: Model Architecture

The input to the model is a student's compilation submission that includes the following:

First, the Critical Code snapshot: the code snapshot that generated the error. Second, the Code History: the most recent code submissions that preceded the critical code submission. We use 4 preceding code submissions as this is the median number of available code snapshots before an error is identified, in both datasets[1]. If the code history was shorter, we used zero-based representations for the empty submissions.

Third, we added four meta features relating to the user. These include: (a) The number of compilation submissions performed before the error occurred. Represents how long the user is working on this program. A higher number of submissions may indicate a struggling or hesitant student. (b) A binary value indicating whether the student generated this error before in any of their previous submission in blueJ. If the student had seen this error before, it may be easier for them to solve this error. (c) A binary value indicating whether the compiler has detected additional errors at the same compilation. Multiple errors may indicate that the student is struggling and will need a longer time to fix the designated error. (d) A value indicating the user experience in the system. For BlueJ, this is the time since the user created the account (available only on BlueJ). For CodeWorkout this is the number of assignments the user has submitted out of the total assignments given in the university course used for the dataset.

We note that if a code submission generated multiple errors, we created multiple instances, one for each error type generated by this code submission. Additionally, we have tried meta-features b,c and d as integers and as binary values and used the representation that had the best results.

[1]We explore the sensitivity of the results to the code history length in Appendix A.

Embedding layer. The first model's layer encodes student's code submission and the history of previous code submissions. We use CodeBert [13] for this embedding layer. CodeBert is a bimodal pre-trained language model for programming languages and natural language text comments. It is based on the RoBERTa-base [22] model architecture, a BERT-based language model with 12 transformer layers, 768-dimensional hidden states, and 12 attention heads. The input format to CodeBert is concatenating two data segments with a special separator token, namely [CLS]. The first segment is natural language text representing the comments in the program and the second segment is the programming code itself. The output of CodeBert includes a representation of the [CLS] token which works as the aggregated representation for the input code snapshot. This output is a 768-dimensional vector and is passed from the embedding layer to the next layer. We note that the maximum input sequence length of CodeBert is 512 tokens. Longer snapshots are truncated to the first 512 tokens[2].

Time-Aware layer. The time-aware layer is designed to reflect the changes in the user code over time. This layer utilizes a Bidirectional Long Short Term Memory (LSTM) [17]. Each code snapshot representation from the previous layer is concatenated to additional four features about the user added to the end of the code snapshot representation. This results in a 772-dimensional vector fed into the Bidirectional LSTM layer. The output of the Bidirectional LSTM layer is a 1544-dimensional vector.

Classification layer. The last layer is the classification layer designed to predict the binary bug-fix-time value (slow or fast). This layer takes the output of the LSTM layer and feeds it into the following layers: (a) A fully connected layer with an output size of 128. This fully connected layer multiplies the input by a weight matrix and then adds a bias vector, using the relu activation function (2) A fully connected layer with an output size of 2 and (3) A Sigmoid function. The output is a binary prediction score of slow or fast time-to-fix.

3.4 Baselines

We evaluated our model against alternative approaches that vary in how students' code is represented and whether the history of past code compilations is considered.

Halstead Metrics Based Method. This baseline is the Halstead metrics method used for measuring code. The method views a computer program as a collection of operator and operand tokens and proposes 12 metrics as described in [16]. In this baseline, we represent each code snapshot with a 12-dimensional vector based on Halstead metrics. This embedding replaces the CodeBert embedding in figure 2. The LSTM in this method is fed with a 16-dimensional vector for each snapshot.

[2]We explore the sensitivity to other truncation approaches in Appendix A.

Code2Vec Based Method. This baseline used Code2Vec [2], an Abstract Syntax Tree (AST) [27] embedding model for code. We used the pre-trained model of Code2Vecv from [2]. The Code2Vec embedding replaces the CodeBert embedding in figure 2.

Critical Code only. This baseline used a version of our proposed model which does not consider past snapshots of the programmer's evolving code. For this baseline, only the critical code submission and the 4 additional meta-features for this submission are used and the LSTM layer is removed.

4. EXPERIMENTS

To address the research questions, four different methods were compared during the experiments:

Halstead Metric Based Method: predicts bug fix time using the code snapshot that contains the error and four preceding snapshots (i.e. code history). Each code snapshot is represented using the Halstead metrics and 4 additional user features (used for RQ1).
Code2Vec Based Method: predicts bug fix time using the snapshot that contains the error and four preceding snapshots. Each code snapshot was embedded using Code2Vec and 4 additional user features (used for RQ2).
Critical code only: predicts bug fix time using the full code snapshot that contains the error and additional 4 features (used for RQ3).
Proposed model: predicts bug fix time using the snapshot that contains the error and four preceding snapshots, each one embedded using CodeBert and 4 additional meta-features.

Our experiments evaluate the above approaches in two different setups: (1) Error-specific: in which a prediction model is trained and evaluated for each error type in separation, and (2) Error-agnostic: where one prediction model is trained and evaluated for multiple error types.

Unfortunately, the CodeWorkout dataset contains on average only 455 instances per error type, so there is not enough data for the error-specific approach for this dataset. Thus, only the error-agnostic model was evaluated for this dataset. All experiments were evaluated using a 5-Fold cross-validation setup and the recommended hyperparameters values from the literature.

The metrics used include: (1) ROC-AUC: summarizes how well the model separates the positive and negative samples for different thresholds. (2) Recall (positive samples): the ratio of positive samples correctly classified as positive to the total number of positive samples. (3) F1 (positive samples): combines the precision (i.e. number of true positive results divided by the number of all positive results) and recall of a classifier into a single metric by taking their harmonic mean.

We focus on the positive samples which represent struggling students that took a long time to fix a bug. Therefore, recall and F1 metrics are measured and reported for this class.

Statistical significance was tested for all results using the Wilcoxon signed rank test [30]. Post-hoc corrections for statistical tests were performed using the Holm-Bonferroni

Table 2: BlueJ - Error Specific Results

Results for Error I			
Method	***Recall[%]***	***F1*[%]**	***ROC-AUC[%]***
Halstead Metric Based	44	49	55
Code2Vec Based	57	56	56
Critical Code Only	64	59	55
Proposed model	**74***	**64***	**62***

Results for Error O			
Method	***Recall[%]***	***F1*[%]**	***ROC-AUC[%]***
Halstead Metric Based	49	52	56
Code2Vec Based	59	56	54
Critical Code Only	57	55	53
Proposed model	**75***	**63***	**60***

Results for Error B			
Method	***Recall[%]***	***F1*[%]**	***ROC-AUC[%]***
Halstead Metric Based	44	46	49
Code2Vec Based	53	50	49
Critical Code Only	63	56	51
Proposed model	**83***	**64***	**54***

Results for Error A			
Method	***Recall[%]***	***F1*[%]**	***ROC-AUC[%]***
Halstead Metric Based	42	46	50
Code2Vec Based	55	52	51
Critical Code Only	60	57	55
Proposed model	**70***	**64***	**61***

method [1]. A star mark ("*") in the results tables (tables 2, 3, 4) denotes a model is significantly better than the rest.

4.1 Error-Specific Results

Table 2 displays the results for error-specific models in BlueJ. As seen in the table, the proposed model outperforms all other approaches for all error types and for all measured metrics. Interestingly, the Code2Vec baseline did not improve over the Halstead metric method in some of the error types on ROC-AUC metric.

4.2 Error-Agnostic Results

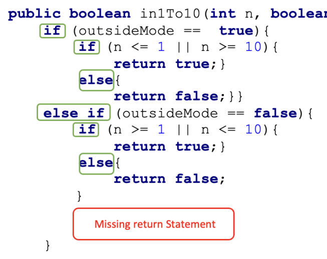
We compare models' performance in the error-agnostic case:

BlueJ Dataset. On the BlueJ dataset, we combined the four errors A, B, I and O. The dataset contained 80,000 instances (sampled from the entire dataset). For each instance in the dataset, the binary labels were determined separately for each error type. Table 3 presents the performance of this dataset. As seen in the table, the proposed method outperformed all baselines in all measured metrics. The second performing approach was the Critical Code Only approach. Code2Vec embedding showed better results than the method based on the shallow Halstead metrics embedding [3].

[3]We evaluate feature importance for the proposed model in Appendix B.

Table 3: BlueJ - Error Agnostic Results

Results for Errors A+B+I+O			
Method	***Recall[%]***	***F1*[%]**	***ROC-AUC[%]***
Halstead Metric Based	47	48	48
Code2Vec Based	55	54	52
Critical Code Only	61	56	55
Proposed model	**77***	**64***	**62***



```
public boolean in1To10(int n, boolean outsideMode){
    if (outsideMode ==  true){
        if (n <= 1 || n >= 10){
            return true;}
        else{
            return false;}}
    else if (outsideMode == false){
        if (n >= 1 || n <= 10){
            return true;}
        else{
            return false;
        }
        Missing return Statement
    }
```

Figure 3: Correct Prediction Using Code Text

CodeWorkout Dataset. For the CodeWorkout dataset, we combined code submissions for all 75 error types into one dataset that contained 80,013 instances. The binary labels were determined based on the median bug-fix-time threshold for each error type in separation. Table 4 presents the results for this dataset. As seen in the table, the proposed model outperformed all other baselines on all measured metrics. The second performing model was the Critical Code Only model. For this dataset, the Code2Vec-based model outperformed the Halstead-based model in 2 of the 3 metrics.

Table 4: CodeWorkout - Error Agnostic Results

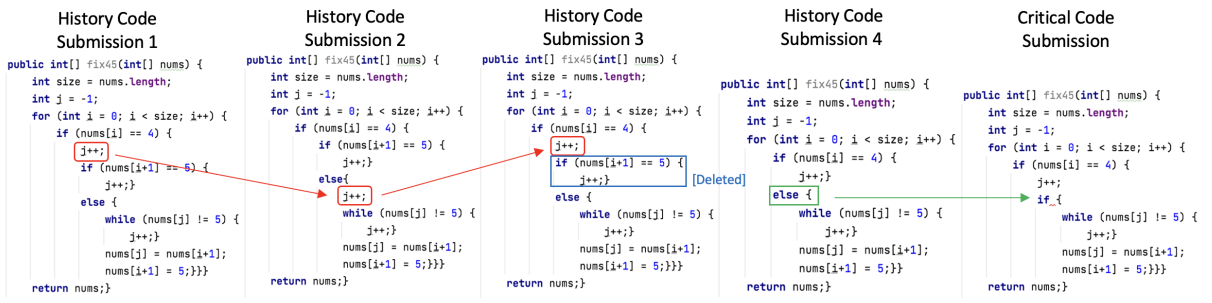
Method	***Recall[%]***	***F1*[%]**	***ROC-AUC[%]***
Halstead Metric Based	54	55	52
Code2Vec Based	58	55	56
Critical Code Only	65	62	65
Proposed model	**70***	**64***	**70***

5. CASE STUDIES

To further demonstrate the performance of the proposed method, we present two illustrative examples.

5.1 Case A: The Value of Code Text

Figure 3 presents a code submission that contains the error "Missing Return Statement". The user generating this error took a long time to fix. While the model that used the Halstead metrics was wrong in predicting a "short" label for this snapshot, the two CodeBert-based models performed a correct prediction. As seen in the figure, the submitted code contains multiple if-else statements which may make it difficult for the student to identify that yet another return statement is missing and its location. We hypothesize that a code-based model correctly classified this sample since it is


History Code Submission 1
History Code Submission 2
History Code Submission 3
History Code Submission 4
Critical Code Submission
[Deleted]

```
public int[] fix45(int[] nums) {
    int size = nums.length;
    int j = -1;
    for (int i = 0; i < size; i++) {
        if (nums[i] == 4) {
            j++;
            if (nums[i+1] == 5) {
                j++;}
            else {
                while (nums[j] != 5) {
                    j++;}
                nums[j] = nums[i+1];
                nums[i+1] = 5;}}}
    return nums;}
```

```
public int[] fix45(int[] nums) {
    int size = nums.length;
    int j = -1;
    for (int i = 0; i < size; i++) {
        if (nums[i] == 4) {
            if (nums[i+1] == 5) {
                j++;}
            else{
                j++;
                while (nums[j] != 5) {
                    j++;}
                nums[j] = nums[i+1];
                nums[i+1] = 5;}}}
    return nums;}
```

```
public int[] fix45(int[] nums) {
    int size = nums.length;
    int j = -1;
    for (int i = 0; i < size; i++) {
        if (nums[i] == 4) {
            j++;
            if (nums[i+1] == 5) {
                j++;}
            else {
                while (nums[j] != 5) {
                    j++;}
                nums[j] = nums[i+1];
                nums[i+1] = 5;}}}
    return nums;}
```

```
public int[] fix45(int[] nums) {
    int size = nums.length;
    int j = -1;
    for (int i = 0; i < size; i++) {
        if (nums[i] == 4) {
            j++;}
        else {
            while (nums[j] != 5) {
                j++;}
            nums[j] = nums[i+1];
            nums[i+1] = 5;}}
    return nums;}
```

```
public int[] fix45(int[] nums) {
    int size = nums.length;
    int j = -1;
    for (int i = 0; i < size; i++) {
        if (nums[i] == 4) {
            j++;
            if {
                while (nums[j] != 5) {
                    j++;}
                nums[j] = nums[i+1];
                nums[i+1] = 5;}}}
    return nums;}
```

Figure 4: Correct Prediction Using the Code History

using the entire code structure, while a shallow model relying only on item counting is blind to such subtle differences.

5.2 Case B: The Value of Code History

Figure 4 presents code submissions that contain an error that brackets are missing with a binary label of "long". While the model that used only the last snapshot (Critical Code Only model) predicted a "short" fix time, the proposed model predicted correctly that the fix time will be "long". Looking at the code submission history may explain why the student is struggling and why it took them a long time to solve the error. As shown in code submissions 1-3 in red, the student changed the code and then changed it back. In code submission 4 they then deleted a full "if" statement and changed an "else" statement to an "if" statement which led to the error. This behavior is most likely a behavior of a struggling student and may indicate that when faced with a resulting error, it will take them a long time to fix it. Such inference can only be captured by a model which considers multiple snapshots and tracks the student's behavior over time.

6. DISCUSSION

The results of this study demonstrate that deep language models built for code representation can significantly improve on past models when predicting bug-fix time (RQ2). They also show that using multiple code snapshots further improves such results (RQ3) validating the benefits of combining the latest language models built for code representation with a multi-snapshot approach. These results reflect the representation power of the latest language models, which are pre-trained on vast amounts of past data. Interestingly, the simpler Halstead method-based approach outperformed the Code2Vec approach in some cases (RQ1). This demonstrates that earlier deep learning methods (such as Code2Vec), which were trained on fewer data and with less sophisticated neural network structures, lack the power inherited in the latest approaches. The key takeaway of these findings is the potential and importance of harnessing such latest language models in the educational data mining field, as it relates to software education and beyond.

We mention some limitations of the proposed approach. First, the computed bug fix time is only an estimation of the true, latent value of the actual time spent by a user on fixing a bug. Even when sign-in and sign-out information is available, such as in the BlueJ dataset, the user may have been occupied with other activities when logged in, contrary to our assumptions. Second, the model assumes each error is independent even though one error can lead to another error. Third, the CodeBert model, similar to other Bert-based models, is limited to 512 input tokens. We used truncation approaches to accommodate this limitation. Nonetheless, future work may consider other approaches (such as summarization, hierarchical representation, etc.) to accommodate longer code snapshots. Fourth, the rapid improvement in language models for code representation implies that CodeBert is only an early bird among an increasing number of evolving models in the field [31]. As such, additional latest models should be investigated in future work.

7. CONCLUSION AND FUTURE WORK

This work provides a new approach for predicting whether a student's bug-fix-time will be "short" or "long" based on a given threshold for common errors made by novice programmers. Predicting a "long" bug-fix-time is one possible way to identify struggling students in need of teacher support. We developed and compared four approaches towards this task (1) A model using Halstead metrics computed over multiple code snapshots preceding a software error (2) A model using Code2Vec for code embedding that considers the code compilation which produced the error and previous student's code snapshots (3) A model using CodeBert for code embedding which considers only the code compilation which produced the error (4) Our approach: a model using CodeBert for code embedding which considers the code submission producing the error and previous student's code submissions. Our approach was able to outperform all baselines for ROC-AUC, Recall, and F1. Our results demonstrate the efficacy of CodeBert and of using multiple time-based code snapshots in identifying struggling students by predicting the bug-fix-time of their software errors.

In future work, we intend to cover additional common student errors and extend this study to different programming languages. Furthermore, during data pre-processing, we found out that some errors are not solved by some students altogether and we plan to extend our model to identify such cases. Finally, we are working on developing and evaluating a regression-based model to predict a continuous bug-fix-time value to better estimate how long it will take students to solve their programming errors.

8. ACKNOWLEDGEMENTS

This work was supported in part by the European Union Horizon 2020 WeNet research and innovation program under grant agreement No 823783.

9. REFERENCES

[1] H. Abdi. Holm's sequential bonferroni procedure. *Encyclopedia of research design*, 1(8):1–8, 2010.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[3] B. S. Alqadi and J. I. Maletic. An empirical study of debugging patterns among novices programmers. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*, pages 15–20, 2017.

[4] A. Altadmri and N. C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527, 2015.

[5] P. Ardimento and C. Mele. Using bert to predict bug-fixing time. In *2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS)*, pages 1–7. IEEE, 2020.

[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, 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] P. Bayman and R. E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Communications of the ACM*, 26(9):677–679, 1983.

[9] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 223–228, 2014.

[10] M. L. Collard, M. J. Decker, and J. I. Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*, pages 516–519. IEEE, 2013.

[11] Y. Dong, S. Marwan, P. Shabrina, T. Price, and T. Barnes. Using student trace logs to determine meaningful progress and struggle during programming problem solving. *International Educational Data Mining Society*, 2021.

[12] 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.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[14] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, 2008.

[15] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56, 2010.

[16] M. H. Halstead. Natural laws controlling algorithm structure? *ACM Sigplan Notices*, 7(2):19–26, 1972.

[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[18] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43, 2007.

[19] 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.

[20] T. Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4, pages 53–58. Citeseer, 2002.

[21] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[22] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[23] Y. Mao, Y. Shi, S. Marwan, T. W. Price, T. Barnes, and M. Chi. Knowing" when" and" where": Temporal-astnn for student learning progression in novice programming tasks. *International Educational Data Mining Society*, 2021.

[24] D. McCall and M. Kölling. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8. IEEE, 2014.

[25] L. D. Panjer. Predicting eclipse bug lifetimes. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 29–29. IEEE, 2007.

[26] Y. Shi, Y. Mao, T. Barnes, M. Chi, and T. W. Price. More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code. *International Educational Data Mining Society*, 2021.

[27] K. Slonneger and B. L. Kurtz. *Formal syntax and semantics of programming languages*, volume 340. Addison-Wesley Reading, 1995.

[28] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pages 3319–3328. PMLR, 2017.

[29] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 1–1. IEEE, 2007.

[30] R. F. Woolson. Wilcoxon signed-rank test. *Wiley*

encyclopedia of clinical trials, pages 1–3, 2007.

[31] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[32] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan. An empirical study on factors impacting bug fixing time. In *2012 19th Working conference on reverse engineering*, pages 225–234. IEEE, 2012.

[33] H. Zhang, L. Gong, and S. Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1042–1051. IEEE, 2013.

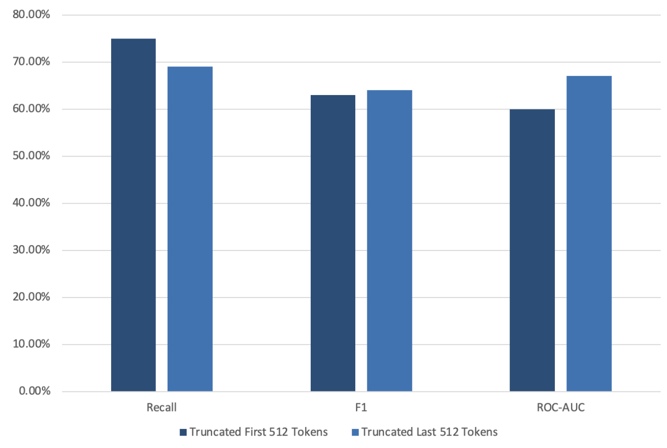
APPENDIX

A. SENSITIVITY ANALYSIS

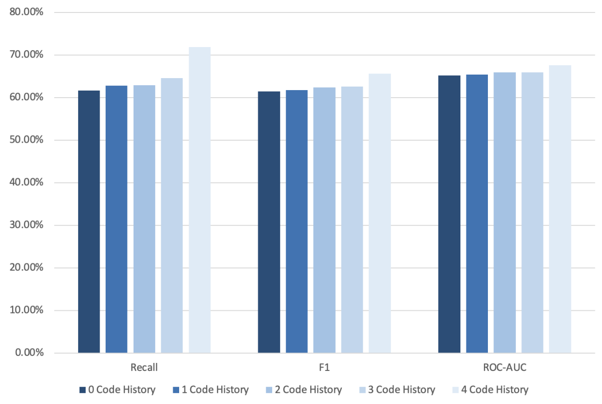
In this section, we analyze the sensitivity of the model to code truncation and the length of snapshot history.

A.1 Code Truncation

As explained in the Embedding layer section, code truncation is needed for some samples to accommodate to CodeBert's (and Bert's) limitation of 512 tokens. This is important for the BlueJ dataset since it contains 57.4% samples over 512 tokens, compared to the CodeWorkout dataset which contains only 1.6% samples over 512 tokens. In this analysis, we compare truncation to the first 512 tokens vs truncation to the last 512 tokens. Figure 5a presents the result of a sensitivity analysis on the BlueJ dataset. As seen in the figure, the result of truncating the first 512 tokens and the last 512 tokens are similar with a slight advantage to the first 512 tokens truncation. Thus, we decided to truncate the code to the first 512 tokens.



(a) Sensitivity Analysis: Code Truncation Approach



(b) Sensitivity Analysis: Code History Length

Figure 5: Sensitivity Analysis

A.2 Code History Length

Figure 5b presents an analysis of the model's performance when manipulating the number of preceding code snapshots included. Specifically, we change the number of such snapshots from 0 to 4 and present the Recall, F1, and ROC-AUC results for these manipulations on the CodeWorkout dataset. As seen in the figure, the model results improve in all metrics as we add more preceding snapshots. Even though the ROC-AUC metric improves only by 2.42% when we move from zero snapshots to 4 snapshots, the recall metric that represents how well the model detected struggling students improve by 10.2%. This suggests not only that history helps to predict bug-fix-time, but that adding more history may improve the performance of the model, pending on the availability of such data.

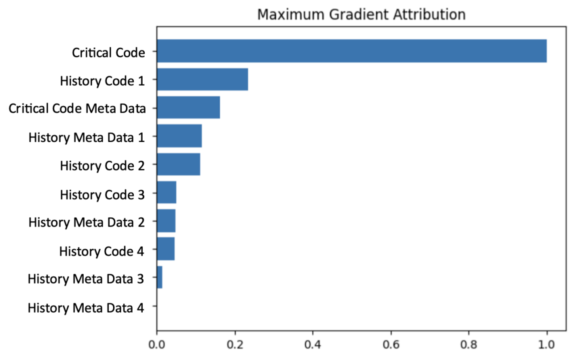


Figure 6: Feature Importance

B. FEATURE IMPORTANCE

To evaluate feature importance, we used Integrated Gradients [28] to calculate feature attribution for the proposed model on the BlueJ dataset. Integrated Gradients are an explainability technique for deep neural networks that visualizes the input feature importance by computing the gradient of the model's prediction output to its input features. In this analysis, we were specifically interested in comparing the importance of different snapshots (latest vs earliest) and the importance of code vs metadata information. To this end, we computed the maximal integrated gradient value for each snapshot and for each metadata group. Calculating the maximum value of each feature group indicates the strongest attribution generated by the group on the output result.

Figure 6 presents the normalized 10 top max attributions. As can be seen in the figure, the critical code snapshot holds the strongest importance, followed by the code snapshot which precedes the critical snapshot (History Code 1). These are then followed in importance by the metadata information from the Critical and History 1 code submissions. Of lower importance are the older code snapshots (History Code 2 and History Code 3). The metadata of Code 2 and Code 3 snapshots and the information of the oldest snapshot (Snapshot 4) are last in line. These results indicate the value captured by both the code itself and the additional metadata information, as well as the value of the information captured from all available historical snapshots (although decreasing as we get earlier in time).