Predicting Defective Visual Code Changes in a Multi-Language AAA Video Game Project

Kalvin Eng
Quality, Verification & Standards
Electronic Arts
Edmonton, Canada
kalvin.eng@{ualberta.ca, ea.com}

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Canada
abram.hindle@ualberta.ca

Alexander Senchenko
Quality, Verification & Standards
Electronic Arts
Vancouver, Canada
asenchenko@ea.com

Abstract—Video game development increasingly relies on using visual programming languages as the primary way to build video game features. The aim of using visual programming is to move game logic into the hands of game designers, who may not be as well versed in textual coding. In this paper, we empirically observe that there are more defect-inducing commits containing visual code than textual code in a AAA video game project codebase. This indicates that the existing textual code Just-in-Time (JIT) defect prediction models under evaluation by *Electronic Arts* (EA) may be ineffective as they do not account for changes in visual code. Thus, we focus our research on constructing visual code defect prediction models that encompass visual code metrics and evaluate the models against defect prediction models that use language agnostic features, and textual code metrics. We test our models using features extracted from the historical codebase of a AAA video game project, as well as the historical codebases of 70 open source projects that use textual and visual code. We find that defect prediction models have better performance overall in terms of the area under the ROC curve (AUC), and Mathews Correlation Coefficient (MCC) when incorporating visual code features for projects that contain more commits with visual code than textual code.

Index Terms—visual code, defect prediction, software quality

I. INTRODUCTION

Visual code, also known as block-based code, low code, no-code, or visual scripts, is a type of software development practice that aims to simplify source code for non-traditional programmers by representing code visually using drag and drop nodes connected via edges, instead of using only text. Non-traditional programmers are end-users who are not well-versed in writing in line-by-line textual code [1]. Visual code, illustrated in Figure 1, attempts to simplify the software development process for non-traditional *end-user programmers* by making dataflow clear and avoiding some classes of syntax errors. Traditional *professional programmers*, in comparison, mainly write software with textual code, and have adequate tool support.

At *EA* (Electronic Arts), video game development teams have shifted towards using visual code as the primary way to build video game features, such as triggers on maps, or game rules, so that all parts of a team can be involved in the software development process. A video game development team is diverse and includes managers, developers, designers, artists, and testers. With many stakeholders, the goal of visual
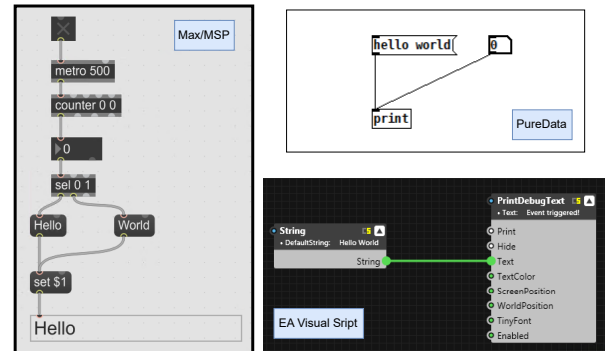

Fig. 1: Visual code examples: Max/MSP, Pure Data, and EA Visual Script.

code is to allow simple changes to code such as using different variables or equations, without waiting for or consulting with traditional *professional programmers*. Popular implementations of visual code occur in game development include Unreal Engine Blueprints [2], and Unity Visual Scripting [3].

Due to the introduction of non-traditional *end-user programmers* using visual code in video game projects, we wish to investigate how to predict defects in visual code. Using the development history of a AAA video game project (a big budget game from a large studio) along with 70 open source visual programming projects to build defect prediction models, we answer the following research questions:

RQ1. Do defect-inducing commits with visual code occur more frequently than defect-inducing commits with textual code in the projects studied?

RQ2. Do visual code features significantly improve the performance of defect prediction models for textual/visual code projects?

RQ3. Do the types of files in commits or choice of learners in projects significantly affect the outcome of defect prediction models?

The goal of this research is to investigate whether contextual features that describe visual code improve defect prediction in projects that use textual and visual code. Visual code metrics, to the best of our knowledge, have never been used for

predicting the likelihood of defects.

II. Background

Defect prediction models are classification models that estimate the likelihood of a change in code being defective [4]. Tantithamthavor *et al.* [4] explain that defect prediction models have been used to predict the likelihood of defects at different granularities including: packages [5], components [6], modules [7], files [5, 8], and methods [9, 10]. Defect prediction models have also been used at version control commit-level [11–13]. The use of defect prediction models at the time of code commits can be referred to as *Just-In-Time* (JIT) defect prediction and can be leveraged immediately once a change is committed to a repository [14]. JIT defect prediction is often used to better understand where to deploy testing efforts [14]. EA uses JIT defect prediction models to perform focused code reviews and testing efforts on specific changes of code at a commit level [15]. The focus of this research is to improve current EA JIT defect prediction models by including features that involve visual code metrics in addition to process metrics, and textual code metrics.

Process metrics capture information about changes during the software development process and aims to describe the relationship between changes and software quality [4]. It is language agnostic and can be applied uniformly to software written in different languages [16]. Examples of process metrics may involve: number of revisions for a file, number of developers for a file, number of modified lines, and number of directories [16, 17]. According to Majumder *et al.* [16], most studies investigating the use of process metrics show that models that only use process metrics can outperform those that combine process metrics and code metrics. However, Kamei *et al.* [5] finds that combining process metrics and code metrics improves defect prediction performance.

Code metrics, also referred to as product metrics, capture information about code and aims to describe the relationship between code properties and software quality [4]. Examples of code metrics can include: lines of code in a file, cyclomatic complexity of code in a file, and number of methods in a file [16–18]. In this paper, we separate code metrics into 2 categories: *textual* which refers to the metrics described in this paragraph that are derived from textual code and *visual* which refers to metrics derived from visual code.

We define visual code metrics as metrics that capture information about visual code to describe the relationship between visual code properties and software quality. Visual code metrics have been investigated by Kumar *et al.* [19] who propose metrics derived from the operators and operands of visual source code defined by the IEC 61131-3 Programmable Logic Controller programming languages standards. Using the same programming languages, visual code metrics have also been investigated by Fischer *et al.* [20] who propose metrics that measure complexity based on size, data structure, control flow, information flow and lexical structure derived in the textual and visual source code. The goal of this investigation

is to derive a suite of visual code metrics suitable for defect prediction in visual code changes.

A. Choices in JIT Defect Prediction

Defect prediction is all about choices that can impact the outcomes of defect prediction models. Since we are concerned about *Just In Time* (JIT) defect prediction, we outline the process summarized in Zhao *et al.* [21] that includes choices in (1) data acquisition, (2) data preparation, (3) model building, and (4) model evaluation.

(1) Data acquisition concerns the sources of data for a defect prediction model which includes retrieving software change history from version control systems and classifying the software changes as defect-inducing or clean [21]. To mark a software change as defect-inducing, defect-fixing changes can be identified from issue tracking systems or commit messages. Defect-fixing changes can be linked to defect-inducing changes using variants of the SZZ algorithm [22, 23]. The choice of how defect-inducing changes are found can affect the outcomes of a defect prediction model [24, 25].

(2) Data preparation encompasses feature acquisition and processing, that is acquiring the sets of metrics from the change history data that will be used for the defect prediction model [21]. Features can be acquired from numerous sources including: software changes, commit messages, issue tracking systems, static analysis, and by automatically learning using algorithms like deep learning [21]. Furthermore, features can be extracted at different levels of granularity depending on what granularity of defect prediction a model will be (e.g. file-level vs commit-level). As well, these extracted features need to be preprocessed including dealing with skew, collinearity and multicollinearity, and class imbalance [21]. Hence, there are many choices to be made about which features to include in a model, and how to preprocess data for building the model.

(3) Model building includes deciding on the granularity of defect prediction, choice of learner(s), and what data to use [21]. The granularity of a defect prediction model is a decision that impacts how interpretable a prediction is. A coarse granularity level would mean prediction at a commit-level, while a fine granularity level would be prediction at a file-level. Wan *et al.* [26] mentions the conclusion of [27] that finds the "practical value of prediction decreases as the granularity level increases", i.e., opting for a coarser granularity may be more practical as it reduces the necessity of reviewing a substantial number of files.

The choice of a learner can impact defect prediction performance [28], thus multiple learners should be tried. There are many popular choices of learners including: Logistic Regression, Tree-based models (e.g. Random Forest, and C4.5 Decision Tree) and ensemble models (e.g. Random Forest, and XGBoost) which affect the performance of defect prediction results [21].

The choice of data used to build the model is also important which includes many factors such as accounting for concept drift, verification latency, defect types, and imbalanced properties of data [21]. Concept drift refers to how regularities in

the software change data gradually change or shift [21]. To address concept drift, it is suggested that different slices of data should be used for training [29].

Another issue to consider is verification latency which refers to the "lag time between when a defect-inducing change is committed to the [version control system] and identified as such" [21]. An oversampling technique is suggested to address verification latency [30]. Defect types should also be considered since there are defect-fixing changes for "extrinsic" defects that occur externally to the code. If extrinsic defects are used to find defect-inducing changes, it can negatively impact model performance as the changes have nothing to do with the fix [31].

Finally, class imbalance, between classes such as defect-inducing and clean, must also be considered. For example, Jiang *et al.* [32] develop a local defect prediction model for defect prediction as they theorize that different developers exhibit different software change patterns. In a local model, a model is created for a subset of data (e.g. a local model for developers means that each developer will use a different model). In contrast, a global model does not account for subsets of data. It should be noted that local models might under-perform global models in defect prediction [33].

(4) Model evaluation refers to how a model's performance is measured and encompasses the validation method used, the choice of evaluation metrics, and whether or not to evaluate feature importance. Validation methods can include cross-validation, however it is suggested that the defect prediction is time-sensitive and therefore data should be split on time instead [34]. Choices of popular model evaluation metrics include: accuracy, precision, recall, F-measure, AUC, and MCC [21, 35]. Different measures should be considered depending on the needs of a defect prediction model and the importance of measuring misclassifications [36]. Feature importance is another dimension that can be explored, this refers to how important a feature is to explain the predicted outcome of a model. Techniques such as LIME [37], SHAP [38], and PyExplainer [39] can be used for feature explanations [4].

B. Investigating Visual Code Defect Prediction

We develop a commit-level visual code defect prediction model, that supports the ongoing EA evaluation of quality assurance which already uses commit-level textual code Just-in-Time (JIT) defect prediction. To encourage replication of our work, we implement a number of visual code defect prediction models using open source projects. Our choices for our visual code defect prediction models are based on the background in Section II-A. Our *data collection* choices are introduced in Section III. Our category of features is explained in Section IV. Our method for preprocessing the data and training the defect predictors is explained in Section V and evaluated in Section VI answering **RQ1**, **RQ2**, and **RQ3**.

III. DATASETS

To build a defect prediction model for evaluation, we use 2 datasets: a AAA video game project, and an open source
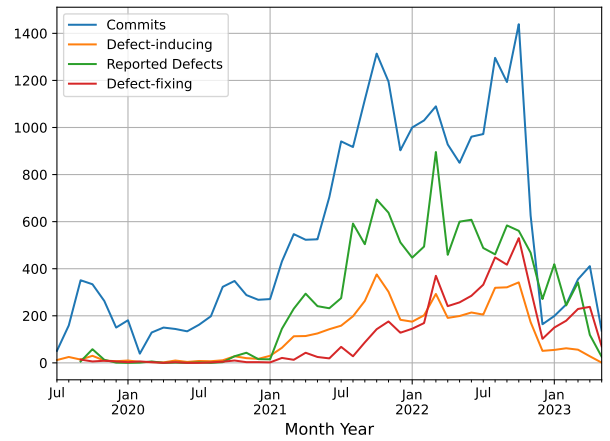

Fig. 2: AAA Video Game project activity over time.

projects dataset. Our AAA video game project is chosen because it is a project that was primarily developed using visual code with 38% of commits consisting of visual code. Our open source projects dataset consists of 70 projects with each project containing a combination of Max/MSP visual code and textual code. The open source projects dataset is used to motivate replication of our work, and to demonstrate generality and feasibility of using visual code features for defect prediction in projects with visual code and textual code.

A. AAA Video Game

In the AAA video game, there are 26,326 commits as of May 14, 2023, where 5,296 are defect-fixing commits, and 5,184 are defect-inducing commits. Defect-inducing commits were found using textual SZZ [40] for any textual code, while SZZ-VC (max change-depth) [23] (an SZZ method for finding defect-inducing changes in visual code) is used for any visual code. To find the defect-fixing changes to identify defect-inducing changes, we consider in the AAA video game project commits that are linked to a fix in the issue tracker. We plot this project activity over time in Figure 2.

This game has been in development since July 2019 and is still actively maintained. The project is forked from a different project that has been in development since March 2014 and uses 399 of its commits. We retrieve our code change commits from Perforce, while issue reports are extracted from Jira. It should be noted that our Jira issue tracker has been designed to work under a quality assurance workflow where defect-fixing commits are recorded in the issue as well as the version of when a defect was first found.

B. Open Source Max/MSP Visual Code Projects

Max/MSP visual code is a popular visual music programming language that is widely used by computer music programmers to realize their music compositions [41]. Burlet *et al.* [41] describe Max/MSP as a language that allow users to programmatically arrange rectangular objects (nodes) on the screen and connect them with lines (edges) called patch cords

to generate sound and respond to human-computer interaction devices. We choose Max/MSP projects because it uses nodes and edges similar to the visual code of the AAA video game project.

To find projects that contain visual code and textual code, we use the U version of the World of Code (WoC) [42] to discover 7,033 initial projects containing a commit in its history with at least one Max/MSP file with the file extension *.maxpat* or *.maxhelp* and includes *"patcher"* in its source code. It should be noted that the projects chosen are the "most central" repository to represent a group of repositories found with the Louvain community detection algorithm in the WoC and does not represent every project in existence on GitHub [43].

We set criteria for projects that we would study:

- They must have at least 200 commits on their main branch (as suggested by Shrikanth *et al.* [44]) after parsing with PyDriller [45];
- Projects must be textual and visual with at least 1 visual commit and 1 textual commit;
- Projects must have at least 1 defect-fixing commit (by matching commit messages using *perfective* keywords from Rosen *et al.* [13]), 1 visual code defect-inducing commit using (SZZ-VC (max change depth) [23]), and 1 textual code defect-inducing commit (using textual SZZ [40]).

As a result, this paper uses **70** open source projects consisting of **64,246** commits (**8,189** defect-fixing commits and **13,002** defect-inducing commits) for evaluating visual code defect prediction.

The **70** projects chosen contain a wide variety of commits with textual code including C, C++, C#, Java, JavaScript, Lua, Objective-C, PHP, Python, Ruby, Swift, and TypeScript along with Max/MSP visual code. This mix of textual and visual code in commits is similar to the AAA video game. The mined open source project data is available for download in our replication package [46].

IV. DEFECT PREDICTION FEATURES

Our study aims to see how visual code metrics affect commit-level defect prediction, hence we choose 3 categories of metrics for features: process metrics, textual code metrics, and visual code metrics. The process metrics are similar to the ones used by Kamei *et al.* [14] and Madeyski *et al.* [17].

A. Process Metric Features

The process metric features are derived from the version control system. We derive our features from valid textual code files and valid visual code files. This derivation of features by file type is similar to the datasets of Ni *et al.* [47] and McIntosh *et al.* [29] which appear to derive metrics for only valid file types. We present each of the process metrics below along with their intuition.

Total Modified File Sizes: By measuring the total size of the code, the larger the file sizes, the greater the potential for defects.

Average Modified File Sizes: By measuring the average size of the code, the intuition is that with larger file sizes, the greater the potential for defects.

Number of Unique Modified Directories [14]: The more directories that a change touches, the greater the potential for defects.

Average Depth of Directories [14]: The deeper the directories in a change, the greater the potential for defects as the change becomes more complex (deeper directories indicate more submodules).

Number of Files Modified [14]: The more files that are modified, the more complex the change and the greater the potential for defects.

Average Elapsed Time Since Last Commit of Modified File [14]: The less time between commits, the greater the potential for defects.

Average Number of Revisions Per File [17]: The more revisions, the greater the potential for defects, because more changes mean that a file is more complex.

Number of Developers [14]: The more developers that have touched a commit, the greater the potential for defects, as each developer can have different ideas about the code.

Number of Unique Changes [14]: The more changes per commit, the more information a developer needs to keep track of, meaning more potential for defects.

Developer Experience [14]: By measuring the number of changes that a developer has made since the beginning of time, more changes indicate more experience and, therefore, a lower likelihood of a defect.

Defect-Fixing [14]: If the change is defect-fixing, then it is unlikely to be a defect.

B. Textual Code Metric Features

These metrics provide context about changes to visual code. We present each of the textual code metrics below along with their intuition.

Total Lines of Code Added [14]: The more lines that are added, the greater the potential for a defect.

Total Lines of Code Deleted [14]: The more lines that are deleted, the more potential there is for a defect.

Total Lines of Code Before Change [14]: The larger a textual code file, the greater the potential for a defect.

Code Entropy [14]: To measure the amount of change across files, we use the same modified lines formula in [14]. The higher the number, the larger the change distributed across many files, implying more potential for a defect.

C. Visual Code Metric Features

The metrics presented below are used to provide context about visual code changes in a commit. The visual code metrics selected are measures that can be used across different visual programming languages, and is meant to represent changes about visual code. We use nodes in-place of lines as visual code is a node and edge based programming language.

Total Number of Nodes Added: The more nodes that are added, the more potential there is for a defect.

Total Number of Nodes Modified: The more modified nodes, the greater the potential for a defect.

Total Number of Nodes Deleted: The more deleted nodes, the greater the potential for a defect.

Number of Nodes Before Change: The larger a visual code program, the greater the potential for a defect.

Node Modification Entropy: Similar to the *Code Entropy* textual code metric, we measure the amount of changes across files by using the code entropy formula in [14] with nodes instead of lines. Larger values indicate a larger distribution of node changes across many files.

V. BUILDING AND EVALUATING DEFECT PREDICTORS

To build defect predictors for the AAA video game, we choose the classic 80/20 non-random split of data where 80% of data is used for training and 20% is used for testing. For the 70 projects of the open source projects dataset, we build predictors for each project and also choose an 80/20 split. Since our split is non-random, it is also time-aware, ensuring that the testing data will never precede any of the training data.

A. Mitigating Collinearity and Multicollinearity

To address collinearity and multicollinearity among features, we use AutoSpearman [48] with the correlation threshold of 0.7 and Variance Inflation Factor of 5 to automatically select our features in the training data.

In the AAA video game project, we find that among the process metrics only the number of developers and whether or not if a commit is defect-fixing are non-correlated. Among the textual code metrics, the number of lines added and the code entropy are non-correlated. For the visual code metrics, the number of nodes added and the node entropy are non-correlated. We use the groups of these features to build our defect predictors for the AAA video game.

In the open source projects dataset, we also automatically select features for each project with AutoSpearman and ensure that there is at least 1 feature in each of the 3 categories (process metrics, textual code metrics, and visual code metrics) outlined in Section IV.

B. Addressing Class Imbalance

To address the class imbalance in the training data of the AAA video game project and open source projects dataset, we apply SMOTE [49] to the training data after mitigating collinearity and multicollinearity among features. The optimal choice for addressing class imbalance is undecided in prior literature, but SMOTE is a widely used technique [21].

C. Learners

We choose a wide variety of learners including ones with interpretability: *Logistic Regression* (LR - Linear Statistical technique), *C5.0* (DT - Decision Tree), and ones that have been demonstrated to be higher performing learners: *Random Forests* (RF - Tree), *XGBoost* (XGB - Tree), *Gradient Boosting Method* (GBM - Tree), *Multi-layer Perceptron* (NN - Neural Network). For each of the learners, we use the default parameters of learners with the `scikit-learn` version 1.2.2 library [50] and xgboost version 1.7.6 library [51].

D. Evaluating Learners and Features for Defect Prediction

We evaluate the effect of defect prediction features by using 6 learners outlined in Section V on 4 different combinations of features: (1) *process metrics only* (Base), (2) *process metrics with textual code metrics* (Textual), (3) *process metrics with visual code metrics* (Visual), and (4) *process metrics with textual code metrics and visual code metrics* (Combined).

Since we train the defect prediction models using the 6 learners for each feature combination, this produces 24 defect prediction models per project. In total there are 24 defect prediction models trained for the AAA video game project, and 1,680 defect prediction models for the open source projects dataset.

We evaluate the performance of each defect prediction model with 2 lenses on performance: *area under the ROC Curve* (AUC) and *Matthews correlation coefficient* (MCC). AUC measures the area under the plot of the true positive rate vs the true negative rate and ranges between 0 and 1, where a 0 indicates that a model is perfectly incorrect, 0.5 indicates that a model is randomly guessing, while a 1 indicates that a model is perfectly correct. MCC measures the correlation between the predicted values and actual values and ranges between -1 and +1, where -1 indicates no agreement, 0 indicates no correlation at all, and +1 indicates perfect agreement. AUC is used for measuring model discriminatory power, while MCC is used for measuring model correctness.

To rank the defect prediction models, we use the *Non-Parametric ScottKnott ESD* (NPSK) test. The NPSK test is a multiple comparison approach that uses hierarchical clustering to partition the set of medians of model performance into statistically distinct ranked groups that are not significantly different or have an insignificant effect size [52, 53]. We use the *ScottKnottESD* R package [54] to perform the test with a 95% significance level. NPSK does not require the assumptions of normal distributions, homogeneous distributions, and the minimum sample size [54].
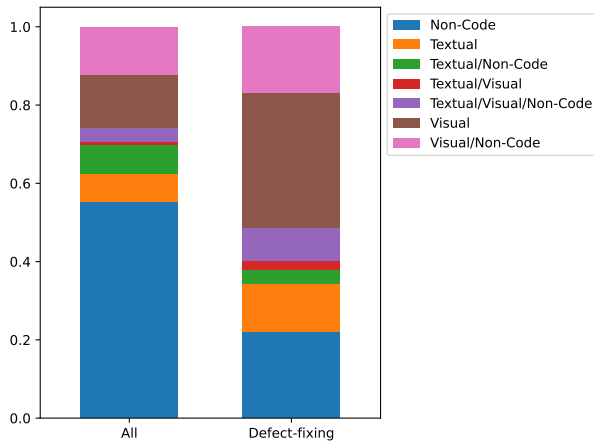
VI. ANALYSIS

RQ1 motivates the need for incorporating visual code metrics into defect prediction models. While **RQ2** investigates if visual code metrics can enhance existing code prediction models, and **RQ3** explores if factors other than visual code metrics can affect the outcomes of defect prediction models.
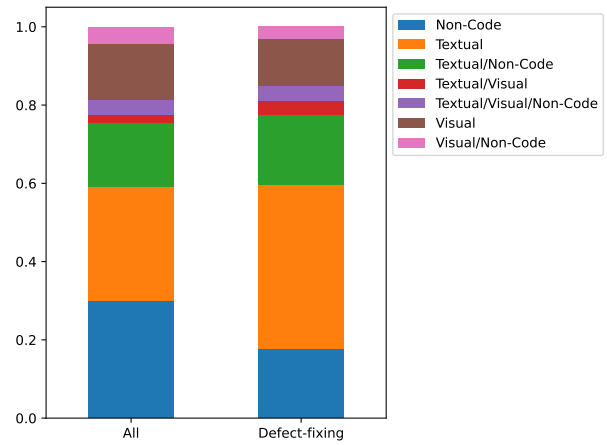
RQ1. Do defect-inducing commits with visual code occur more frequently than defect-inducing commits with textual code in the projects studied?

We refer to *textual code* as any code with a file extension that is able to be processed by the Python Lizard library [55] which contains a subset of the file extensions in Commit Guru [13]. We refer to *visual code* as any code that can be parsed as visual code and contains file extension types related to the visual programming language at EA for the AAA video game and Max/MSP for the open source projects dataset.

There are 7 possible file type combinations in commits: (1) only non-code files, (2) only textual code, (3) only visual code,

(a) AAA video game project.

(b) Open source projects dataset.

Fig. 3: Percentage of commits containing different file types.

(4) textual code and non-code files, (5) visual code and non-code files, (6) textual code and visual code, and (7) textual code, visual code, and non-code files. These are of particular interest because the previous prediction model for the AAA video game only considered the textual combinations (1), (2), (4) meaning that important contextual information about visual code is missed.

We present a stacked bar chart of the file type combination distribution for the 26,326 commits of the AAA video game project in Figure 3a. For all commits, a majority contain only non-code files (55%). However, with code files, there are more visual code files present (38%) in commits than textual code files present (19%) in commits.

To understand what types of defects there are in the AAA video game project, we use the 5,296 defect-fixing commits as a proxy for the defects (i.e. a defect-fixing commit will change at least one defective file, hence it represents at least one defect by proxy). We visualize the breakdown of the 7 possible file type combinations of defect-fixing commits in Figure 3a. We can see that there is a larger number of defect-fixing commits containing visual code (62%) than defect-fixing commits containing textual code (27%). Therefore, we can conclude that visual code defects occur more than textual code defect commits within this project. This motivates our investigation into incorporating visual code for defect prediction models.

For the open source projects dataset, we present an aggregated chart of the file type combinations in commits for the 64,246 commits of the 70 projects in Figure 3b. Overall there are 30% of commits that contain only non-code files, while 51% of commits contain at least one textual code file and 25% of commits contain at least one visual code file. We also look at the file types in defect-fixing commits and find that 67% of commits contain textual code, while only 23% of commits contain visual code. These ratios are different from those seen in the AAA video game project.
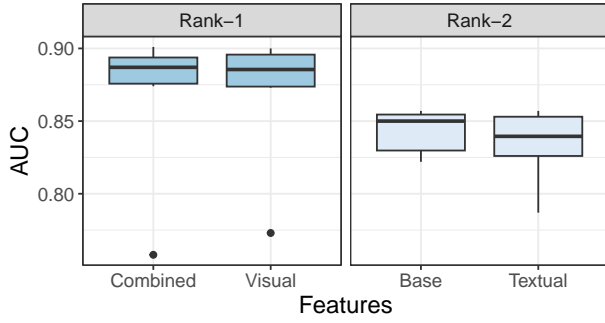
Seeing that the overall ratios of the open source projects dataset is different from the AAA video game project, we also see which of the 70 projects are individually similar to the breakdown of the AAA video game project where there are more commits with visual code files than textual code files. We find that 20 open source projects have more visual code files than textual files in all commits. While we find that 19 open source projects have more visual code files than textual files in defect-fixing commits.

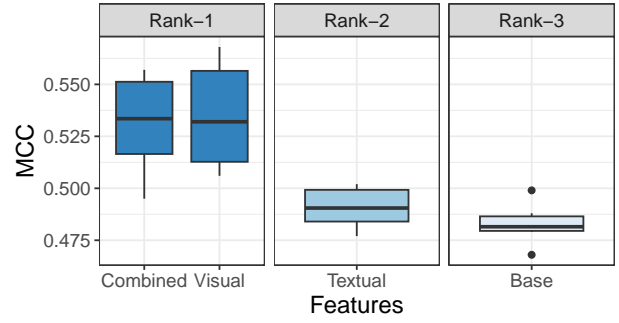> *Defects are occurring in visual code, so they need to be addressed.*

RQ2. Do visual code features significantly improve the performance of defect prediction models for textual/visual code projects?

This RQ is concerned with model performance and the effect of visual features on model performance. To determine if visual code features can improve the performance of defect prediction models, we apply NPSK to the AUC and MCC results of each defect prediction model grouping by the Base, Textual, Visual, and Combined feature combinations described in Section V-D.

We can see the results for the AAA video game project in Figure 4. In Figure 4a, the Combined and Visual feature combinations are ranked higher than the Base and Textual feature combinations for AUC meaning that visual code metrics contribute to a statistically significant difference in model prediction ability for the AAA video game project. We also can conclude that there is a significant difference in MCC, as we can see in Figure 4b that the Combined and Visual feature combinations rank higher than the Base and Textual feature combinations.
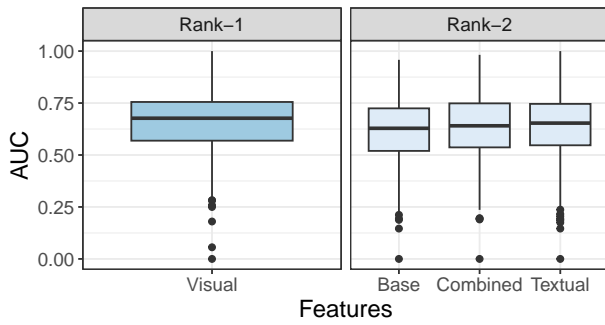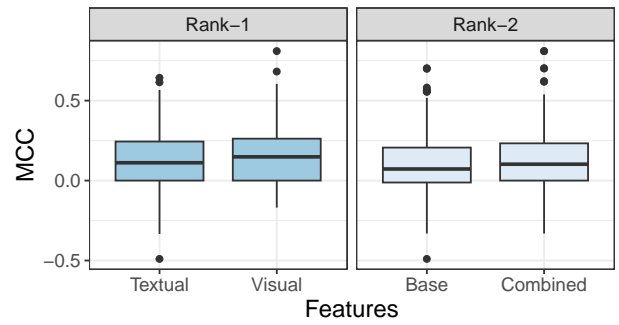
(a) AUC

(b) MCC

Fig. 4: NPSK rankings of feature groups by AUC and MCC in AAA video game project.

(a) AUC

(b) MCC

Fig. 5: NPSK rankings of feature groups by AUC and MCC in open source projects.

The results of NPSK grouping by feature combinations for the open source projects can be seen in Figure 5. NPSK is applied across projects and is used to determine if feature combinations are a significant factor in the group of projects. In terms of AUC, we can conclude that visual code feature combinations can help improve model performance as it ranks higher than the other feature combinations in Figure 5a. In terms of MCC, we can conclude using Figure 5b that using only textual code feature combinations or using only visual code feature combinations can significantly improve model performance (MCC). However, when combining textual code and visual code features together, it will perform only as well as using no textual code and no visual features at all.

Overall, we conclude that the addition of visual code features can improve AUC in both the AAA video game project and the open source projects. We also conclude that using visual code features can improve MCC for both the AAA video game project and the open source projects.
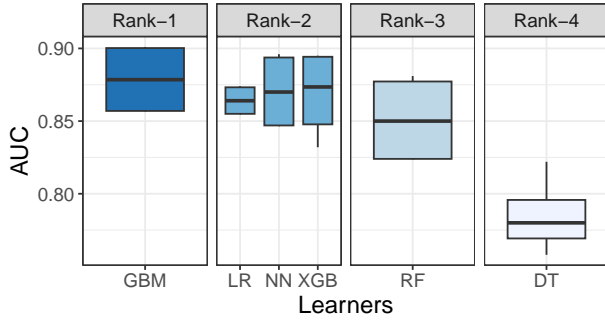
> *Contextual features for visual code improve defect prediction performance.*

RQ3. Do the types of files in commits or choice of learners in projects significantly affect the outcome of defect prediction models?
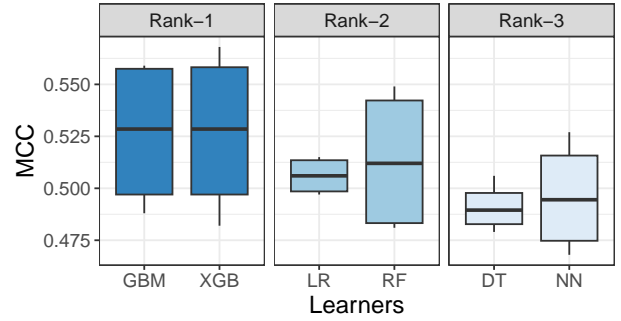
In RQ1, we find that the distribution of visual code files and textual code files in commits of projects can vary. Therefore, the first factor we consider is if projects have more commits containing visual code files or more commits containing textual code files. The second factor we consider is the choice of learners to train the defect prediction models for the Base, Textual, Visual, and Combined combination of features described in Section V-D.

For the first factor, we split the projects into 2 groups where the *first group* consists of 17 projects with more commits containing visual code files than textual code files and the *second group* consists of 53 projects with more commits containing textual code files than visual code files. To determine if there is any significant difference among distributions of the AUC and MCC in the 2 groups, we perform the *Wilcoxon rank-sum test* (WRST). The null hypothesis for the WRST test is that the distributions are not significantly different among each other. We reject the null hypothesis if $p < 0.05$.

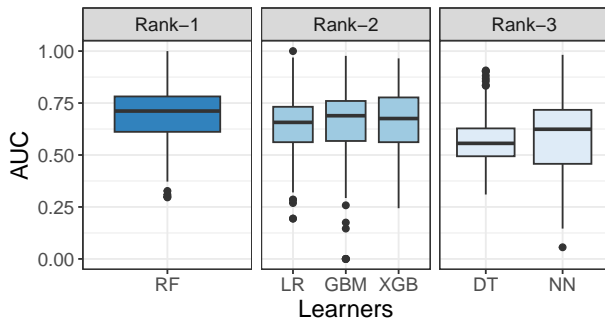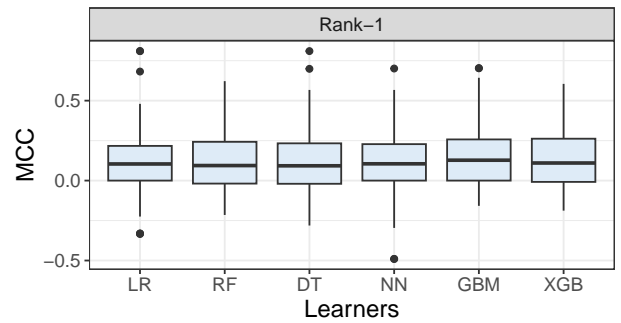With the distributions of MCC in the 2 groups, we reject

(a) AUC

(b) MCC

Fig. 6: NPSK rankings of learners by AUC and MCC in AAA video game project.

(a) AUC

(b) MCC

Fig. 7: NPSK rankings of learners by AUC and MCC in open source projects.

the null hypothesis (p = 0.01 < 0.05) meaning that there is a difference in a model's MCC performance among the 2 groups. With the distributions of AUC in the 2 groups, we do not reject the null hypothesis (p = 0.97 > 0.05) implying that the performance of models produced in each group do not significantly differ. However, there is no definitive conclusion about AUC being affected by majority visual code or majority textual code. Thus, we only conclude how well a model performs in terms of MCC can be affected by majority visual code or majority textual code.

For the second factor, we wish to see if the learners affect the outcomes of prediction models in terms of AUC and MCC. We apply NPSK to the MCC and AUC evaluation of each defect prediction model grouping by learner in Figure 6 for the AAA video game project and in Figure 7 for the open source projects. We can see that the GBM (Gradient Boosting Method) learner model performs the best for both AUC and MCC in the AAA video game project. In the open source projects, we see that the RF (Random Forest) model performs the best for AUC, but the learner does not matter for MCC. From these results, we can conclude that learners do have a weak effect on the model prediction outcome, borne out only in AUC, not MCC.

The types of files in commits, and learner choice matter for model performance, but perhaps not as much as contextual features do.

VII. THREATS TO VALIDITY

Internal validity concerns the quality of labelled defective commits and bug reports used, how the features are chosen and extracted, and how the model is trained and tested which can affect the outcomes of our evaluation. To these extents, we may not have identified all potential defect-fixing commits, especially in our open source projects where we search commit messages for *perfective* keywords. Furthermore, SZZ is imperfect [24, 25] meaning that not all defect-inducing commits may have been identified with our implementations of SZZ.

Construct validity is if our conclusions follow from our assessments. We provide a rationale for our choice of performance measures and explore how file types in commits, and learners can affect model prediction outcomes with RQ2. We make careful use of statistical tools, such as NPSK, that address issues such as multiple hypothesis testing and non-parametric distributions.

External validity considers the extent to which the visual code defect prediction models can be applied to other projects

that use visual programming languages, beyond those we selected for our evaluation. This validity is limited by the lack of open source projects with labelled defects that are similar to the AAA video game project. We attempt to address this by using more open source projects to explore the generality of our conclusions for the AAA video game project, but we are limited to just 2 visual programming languages. External validity is bolstered by the open release of our replication package for open source projects [46]

VIII. CONCLUSION

In this paper, we present the use of visual code metrics for defect prediction to predict if a commit is defect-inducing. This work is motivated by the prevalent use of visual code to develop video game features at EA. We demonstrate how the performance of our current defect prediction models, which only consider process metrics and textual code metrics, could be improved by also including visual code metrics. We also conclude that visual code metrics improve the performance of defect prediction models for many open source projects.

Future work includes investigating contextual visual code features such as dataflow, complexity measures, and code embeddings, to improve visual code defect prediction models.

The results of our research outline how incorporating visual code metrics into defect prediction models can help benefit JIT defect prediction at EA. To improve the replicability of our work, we release a replication package [46] with 70 open source projects that contain visual code and textual code similar to the AAA video game project studied.

REFERENCES

[1] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers *et al.*, "The state of the art in end-user software engineering," *CSUR*, vol. 43, no. 3, pp. 1–44, 2011.

[2] Epic Games, Inc., "Blueprints visual scripting in unreal engine," 2023. [Online]. Available: https://web.archive.org/web/20230316091606/https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/

[3] Unity, "Unity visual scripting," 2021. [Online]. Available: https://web.archive.org/web/20230329001942/https://unity.com/features/unity-visual-scripting

[4] C. Tantithamthavorn and J. Jiarpakdee, *Explainable AI for Software Engineering*. Monash University, 2021, retrieved 2021-05-17. [Online]. Available: http://xai4se.github.io/

[5] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *ICSME*. IEEE, 2010, pp. 1–10.

[6] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *ICSE*, 2016, pp. 1039–1050.

[7] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *ESEM*. IEEE, 2007, pp. 196–204.

[8] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *CSMR*. IEEE, 2010, pp. 107–116.

[9] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *ICSE 2012*. IEEE, 2012, pp. 200–210.

[10] L. Pascarella, F. Palomba, and A. Bacchelli, "On the performance of method-level bug prediction: A negative result," *JSS*, vol. 161, p. 110493, 2020.

[11] ——, "Fine-grained just-in-time defect prediction," *JSS*, vol. 150, pp. 22–36, 2019.

[12] M. Nayrolles and A. Hamou-Lhadj, "Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in *MSR*, 2018, pp. 153–164.

[13] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *ESEC/FSE*, 2015, pp. 966–969.

[14] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *TSE*, vol. 39, no. 6, pp. 757–773, 2012.

[15] A. Senchenko, J. Patterson, H. Samuel, and D. Ispir, "Supernova: Automating test selection and defect prevention in aaa video games using risk based testing and machine learning," in *ICST*. IEEE, 2022, pp. 345–354.

[16] S. Majumder, P. Mody, and T. Menzies, "Revisiting process versus product metrics: a large scale analysis," *EMSE*, vol. 27, no. 3, p. 60, 2022.

[17] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? an empirical study," *SQJ*, vol. 23, pp. 393–422, 2015.

[18] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *TSE*, vol. 31, no. 10, pp. 897–910, 2005.

[19] L. Kumar, R. Jetley, and A. Sureka, "Source code metrics for programmable logic controller (plc) ladder diagram (ld) visual programming language," in *WETSoM*. IEEE, 2016, pp. 15–21.

[20] J. Fischer, B. Vogel-Heuser, H. Schneider, N. Langer, M. Felger, and M. Bengel, "Measuring the overall complexity of graphical and textual iec 61131-3 control software," *RA-L*, vol. 6, no. 3, pp. 5784–5791, 2021.

[21] Y. Zhao, K. Damevski, and H. Chen, "A systematic survey of just-in-time software defect prediction," *CSUR*, vol. 55, no. 10, pp. 1–35, 2023.

[22] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SEN*, vol. 30, no. 4, pp. 1–5, 2005.

[23] K. Eng, A. Hindle, and A. Senchenko, "Identifying defect-inducing changes in visual code," in *ICSME*. IEEE, 2023.

[24] S. Quach, M. Lamothe, B. Adams, Y. Kamei, and

W. Shang, "Evaluating the impact of falsely detected performance bug-inducing changes in jit models," *EMSE*, vol. 26, pp. 1–32, 2021.

[25] Y. Fan, X. Xia, D. A. Da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," *TSE*, vol. 47, no. 8, pp. 1559–1586, 2019.

[26] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, "Perceptions, expectations, and challenges in defect prediction," *TSE*, vol. 46, no. 11, pp. 1241–1266, 2018.

[27] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in *SANER*, vol. 5. IEEE, 2016, pp. 33–45.

[28] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *ICSE*, vol. 1. IEEE, 2015, pp. 789–800.

[29] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," in *ICSE*, 2018, pp. 560–560.

[30] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *ICSE*. IEEE, 2019, pp. 666–676.

[31] G. Rodriguez-Perez, M. Nagappan, and G. Robles, "Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project," *TSE*, vol. 48, no. 4, pp. 1400–1416, 2020.

[32] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *ASE*. IEEE, 2013, pp. 279–289.

[33] X. Yang, H. Yu, G. Fan, K. Shi, and L. Chen, "Local versus global models for just-in-time software defect prediction," *Scientific Programming*, vol. 2019, 2019.

[34] A. A. Bangash, H. Sahar, A. Hindle, and K. Ali, "On the time-based conclusion stability of cross-project defect prediction models," *EMSE*, vol. 25, no. 6, pp. 5047–5083, 2020.

[35] J. Yao and M. Shepperd, "Assessing software defection prediction performance: Why using the matthews correlation coefficient matters," in *EASE*, 2020, pp. 120–129.

[36] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *TSE*, vol. 38, no. 6, pp. 1276–1304, 2011.

[37] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in *KDD*. ACM, 2016, pp. 1135–1144.

[38] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *NeurIPS*, vol. 30, 2017.

[39] C. Pornprasit, C. Tantithamthavorn, J. Jiarpakdee, M. Fu, and P. Thongtanunam, "Pyexplainer: Explaining the predictions of just-in-time defect models," in *ASE*. IEEE, 2021, pp. 407–418.

[40] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *ASE*. IEEE, 2006, pp. 81–90.

[41] G. Burlet and A. Hindle, "An empirical study of end-user programmers in the computer music community," in *MSR*. IEEE, 2015, pp. 292–302.

[42] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus, "World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data," in *MSR*. IEEE, 2019, pp. 143–154.

[43] A. Mockus, D. Spinellis, Z. Kotti, and G. J. Dusing, "A Complete Set of Related Git Repositories Identified via Community Detection Approaches Based on Shared Commits," *MSR*, Jun 2020.

[44] N. Shrikanth and T. Menzies, "Assessing the early bird heuristic (for predicting project quality)," *TOSEM*, 2023.

[45] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories*. ACM, 2018.

[46] K. Eng, A. Hindle, and A. Senchenko, "Replication Package of "Predicting Defective Visual Code Changes in a Multi-Language AAA Video Game Project"," Aug. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8286531

[47] C. Ni, X. Xia, D. Lo, X. Yang, and A. E. Hassan, "Just-in-time defect prediction on javascript projects: A replication study," *TOSEM*, vol. 31, no. 4, 2022.

[48] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "Autospearman: Automatically mitigating correlated software metrics for interpreting defect models," in *ICSME*. IEEE Computer Society, 2018, pp. 92–103.

[49] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *JAIR*, vol. 16, pp. 321–357, 2002.

[50] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *JMLR*, vol. 12, pp. 2825–2830, 2011.

[51] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *KDD16*. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[52] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *TSE*, vol. 43, no. 1, 2017.

[53] ——, "The impact of automated parameter optimization for defect prediction models," *TSE*, 2018.

[54] ——, "The scottknott effect size difference (esd) test (version 3.0, the development branch)," 2023. [Online]. Available: https://github.com/klainfo/ScottKnottESD/tree/development

[55] T. Yin *et al.*, "Lizard," 2023. [Online]. Available: https://github.com/terryyin/lizard