Code Metrics Based Bug Classification using H2O AutoML

Context: Classifying the priority of software bugs is an important activity in software maintenance. Manually screening error reports and determining the priority of each bug is a time-consuming process that requires resources and expertise. In many cases, assigning priorities manually causes errors, and prevents developers from completing tasks, fixing errors, and improving quality Objective: The purpose of this study is to summarize, analyze and evaluate the proposed software bug prediction techniques using ma-chine learning (ML). In addition to provide and evaluate the experimental evidence gained from the past studies regarding ML techniques and bug prediction models. Also, it aims to predict bugs in systems by code metrics using ML and priorities them. Method: Given a UNIFIED dataset of bugs from an open-source soft-ware project, this study uses H2O AUTOML to detect and categories code bugs. The process of assigning the category for code bugs consists of multi phases, which involves data collection, formatting priority of the bugs, features' selection, dataset training, applying H2O algorithm, and finally the evaluation process. The proposed structure involves using H2O AUTOML to determine the best model that matches the bugs unified dataset. Results: The results showed that the best model is "GBM 2 AutoML 20210415 132614" from the Gradient Boosting Machine family with an accuracy of 73% and mean per class error of 46%. Conclusion: This research provides a framework that can automatically assign appropriate classes to code bugs, thereby avoiding time-consuming and resource-limited software testing. The proposed structure involves using the library H2O AUTOML to determine the best model that matches the bugs unified dataset. The ability to use H2O AutoML to classify bugs in Unified dataset has been demonstrated, H2O exceeds expectations in terms of ease of use and scalability, active customer base, and open-source machine learning community.

Keywords: *Bug Prediction, AutoML, Code Metrics, Classification, H2O.*

Introduction

Software bug prediction is a popular research area in the field of software engineering today. Bug or defect prediction is a process by which researchers try to learn from mistakes committed in the past and build a prediction model to leverage the location and number of future bugs.

Finding and eliminating bugs in software systems has always been one of the most important issues in software engineering. If the bugs are detected earlier through prediction, the quality of software can be improved. [2] When bugs are found before the release of the software, they can be removed before the deployment of the software.

The goals of software bug prediction, especially when being applied to the early stage, are to increase the value of the software and lessen the cost, which

eventually offers well-planned software management. There has been impressive development and premium in modern uses of AI (ML) as of late. ML engineers, as a result, are sought after across the business, yet improving the proficiency of ML engineers stays a central test.

Mechanized AI (AutoML) has arisen as an approach to save time and exertion on dull assignments in ML pipelines, for example, information handling, include designing, model determination, hyperparameter streamlining, and forecast result examination. H2O AutoML is a robotized AI remembered for the H2O structure. That is easy to utilize and delivers top-notch models that are appropriate for arrangement in an undertaking climate.

H2O AutoML upholds administered preparing of relapse, double order, and multi-class grouping models on plain datasets. One of the advantages of H2O models is the quick scoring abilities – numerous H2O models can produce expectations in sub millisecond scoring times. H2O AutoML offers APIs in a few dialects (R, Python, Java, Scala) which implies it tends to be utilized flawlessly inside an assorted group of information researchers and specialists [1].

Literature Review

This article focuses on related works and studies that are mainly related to machine learning (ML) and the techniques were applied to software bugs classification. This section introduces recent studies and literature that are related to bugs' classification.

Malhotra [2] focused on the objective of programming bug expectation to distinguish the product modules that will have the probability to get bugs, by utilizing some key undertaking assets before the genuine testing begins. Because of significant expense in revising the recognized bugs. They conducted a survey of past writing on programming bug forecast and AI to understand the suitable way toward building the expectation model. Not just they need to see the AI procedures that previous analysts utilized, they likewise survey the datasets, measurements and execution estimates that are utilized during the improvement of the models. In this investigation, they have limited to 31 fundamental examinations and six kinds of AI procedures have been recognized.

Two public datasets are discovered to be much of the time utilized and object-arranged measurements are the exceptionally picked measurements for the forecast model. Concerning the presentation measure, both graphical and mathematical measures are regularly used to assess the exhibition of the models. From the out-comes, they presume that the AI procedure can foresee the bug, however there are relatively few applications in this space that exist these days.

Tóth Zoltán [3] examined two principal themes, these being the development of new bug datasets with their assessment in bug forecast, and a philosophy for estimating practicality in heritage frameworks written in the RPG programming language. In the event of bug datsets, they gathered existing public bug datasets that utilization static programming item measurements to describe the bugs. These datasets assembled from different sources, like Source-Forge, Jira, Bugzilla, CVS,

and SVN. GitHub, being a pattern for facilitating open-source projects, was a decent contender to accumulate new ventures from.

They developed another dataset to beat these inadequacies and to propose another dataset with exceptional bug information. Also, they introduced a technique for binding together open bug datasets, in this way they share normal measurements as descriptors. They showed how heterogeneous distinctive datasets can be by contrasting their metric suites. They likewise introduced the capacities of assembled bug expectation models on account of the recently made GitHub Bug Dataset and on account of the Unified Bug Dataset also. They recommend that specialists should initially take a stab at utilizing existing bug datasets, and just if none of them adjusts to their necessities, build another redid dataset for their very explicit prerequisites [3].

Abozeed et al. [1] demonstrated that the expense of fixing mistakes raises as task travels through its life cycle in an outstanding style. Recognizing carriage classes, when they are focused on the Form Control System, would essentially affect decreasing such expense. In this paper, tests are done to consider the impact of highlight determination on the presentation of bug forecast models furthermore, to check if better outcomes can be gotten by utilizing the promising Deep Learning procedures. Results show that applying include choice, utilizing a basic channel approach, for example, choosing the exceptionally positioned 9 and 5 highlights out of the 17 highlights, did not improve the exhibition measures as a rule. On the other hand, results show that Deep Learning model (DL) accomplishes better measures than choosing a set of base classifiers for little and adjusted datasets.

Perera [4] introduced SBST methods, replacing costly assignment of physically composing experiments. SBST strategies are viable at producing tests with high code inclusion. They argue that SBST should be engaged to look for experiments in imperfect zones maybe in non-flawed spaces of the code to boost the probability of finding the bugs. Imperfection forecast calculations give valuable data about the bug inclined regions in programming. Subsequently, they aimed to improve the bug discovery ability of SBST by joining deformity expectation data. They devise two examination targets, i.e., 1) Develop a novel methodology SBST that dispenses time spending plan to classes dependent on the probability of classes being deficient, and 2) Foster a novel technique SBST to manage the hidden pursuit calculation (i.e., hereditary calculation) towards the flawed territories in a class. Through exact assessment on 434 genuine revealed bugs in the Defects4J dataset, they exhibit that the novel methodology, SBST, is altogether more effective than the best-in-class SBST at the point when they are given a tight time spending plan in an asset compelled situation.

Other researchers [5] utilized bug datasets to construct and approve novel bug expectation models. They focused on gathering existing public source code metric-based bug datasets and bind together their substance. Moreover, they wish to evaluate the plenty of gathered measurements and the abilities of the brought together bug datasets in bug forecast. They considered 5 public datasets and downloaded the relating source code for every framework in the datasets and performed source code investigation to get a typical arrangement of source code

measurements. Along these lines, they delivered a bound together bug dataset at class and document level also. They examined the redirection of metric definitions and upsides of the diverse bug datasets. At last, they utilized a choice tree calculation to show the capacities of the dataset in bug forecast. They combined all classes (and documents) into one dataset which comprises of 47,618 components (43,744 for records) and assessed the bug prediction model expand on this dataset also. At last, they additionally examined cross-project abilities of the bug prediction models and datasets.

Ferenc et al. [6] investigate growing dependence on programming items and how to discover bugs as ahead of schedule and as effectively as expected. They propose to restore static source code measurements and use them with deep learning – among the most encouraging and generalizable prediction techniques – to find suspicious code fragments at the class level. They show a point-by-point philosophy of how they adjusted deep neural networks, applied them to a bug dataset (containing 8780 messed with and 38,838 not messed with Java classes), and compared them using different metrics. They show that deep learning with static measurements can undoubtedly support prediction correctness's. The best model has a F-score of 53.59%, which increments to 55.27% for the best group model containing a profound learning segment.

Kaen et al. [7] investigate the evolution and development in the field of man-made brainpower and its different branches, for example, Machine Learning (ML) and Deep Learning in different imperative fields like advanced mechanics, smart vehicles, urban communities, medical services, computer programming and numerous different fields. They propose to use the Chi-Square feature selection method to find features importance. They also propose to build ML models by using both of 1) top ten important features and 2) top five important features, based on the three ML classifications algorithms, Support Vector Machine (SVM), Naïve Bayes (NB), and Linear Discriminant Analysis (LDA). Results showed that the proposed approach against baseline achieved an improvements as average accuracy reaching up to 5.12%, 4.15% and 1% on the NB, SVM and LDA classifiers respectively.

Khan and others [8] introduced programming bug prediction (SBP) models to improve the product quality assurance (SQA) measures by predicting buggy components. The bug prediction models use ML classifiers to predict bugs in software parts in some software metrics. Numerous strategies have been proposed by specialists to predict the imperfect segments, yet these classifiers sometimes do not perform well when default settings are utilized for AI classifiers. They use ML classifiers related to the Artificial Immune Network (AIN) to improve bug prediction accuracy through its hyper-boundary streamlining. For this reason, seven classifiers, for example, support vector machine Radial base capacity (SVM-RBF), K-closest neighbor (KNN) (Minkowski metric), KNN (Euclidean measurement), Naive Bayes (NB), Decision Tree (DT), Linear segregate investigation (LDA), Random woods (RF) and versa-tile boosting (AdaBoost), were utilized. The results showed that hyper-boundary improvement of ML classifiers, utilizing AIN and its applications for programming bug prediction, performed better compared to when classifiers with their default hyper-boundaries

were utilized.

Wahono [9] introduced a systematic literature review that aims to identify and analyze the techniques, methods, frameworks, and datasets which are used in software defect prediction. Results showed that researchers proposed many techniques for improving the accuracy of ML classifiers for software defect prediction using different methods, different algorithms, and using different optimization methods for some classifiers.

Pandey et al. [10] propose a simple grouping-based system Bug Prediction utilizing Deep portrayal and Ensemble learning (BPDET) methods for SBP. The product measurements which are utilized for SBP are generally traditional. Marked denoising auto-encoder (SDA) is utilized for the deep portrayal of programming measurements, which is a strong element learning strategy. The proposed model is primarily separated into two phases: deep learning stage and two layers of EL stage (TEL). The analysis was performed using 12 datasets extracted from NASA, to uncover the effectiveness of DR, SDA, and TEL. Different metrics have been used such as: Mathews Correlation Coefficient (MCC), the Area Under Curve (AUC), Precision-Recall Curve (PRC), F-measure, and Time. Out of 12 dataset MCC values over more than 11 datasets, ROC values over than 6 datasets, PRC values over 12 datasets and F-measure more than 8 datasets which outperform the existing studies.

Brumfield [12] focus is to improve the prediction of programming bugs using miniature examples with deep learning techniques. Software bug prediction at a better granularity level will allow designers to restrict code to test and troubleshoot during the development process. Existing research demonstrates that these methods help designers in distinguishing programming that has possible deformities during development. The major goal is to restrict the product bugs to guarantee that only the affected documents are addressed to test and troubleshoot. Researchers have discovered relationships amongst measurements and programming absconds in programming undertakings to construct perceptive models utilizing static examination and AI procedures.

Bilgin and others [13] aim to develop a ML model for vulnerability prediction based on the AST representation of source code, and transfer as much information as possible from AST to numeric array representation. According to the experimental analysis, this process reduces training time by about 68%. The authors first proposed a source code representation method that is capable of characterizing source code into a proper format for further processes in ML algorithms. The authors conducted many experiments under different settings with the objective of predicting five different predetermined vulnerability types and achieved promising and encouraging results compared to state-of-art methods.

Felix [14] demonstrated how certain optimal variables can be applied both in data preprocessing, and in predicting the possible number of defects in a future version of a software product, using information available from the current version at the method level of the software. Also, Trautsch and others [15] combined a state-of-the art just-in-time defect prediction approach with additional static source code metrics from OpenStaticAnalyzer and static analysis warnings from a well-known Java static analysis tool (PMD).

Also, others [16] proposed research that aims to automatically recognize relevant functions, which later should be revised manually by programmers; that is the authors were interested in achieving high recall and trading precision for recall if needed. Machine learning methods were applied to a software engineering problem of fault prediction.

Lee et al. [17] proposed 56 micro interaction metrics (MIMs). They proposed defect prediction models using the MIMs, traditional metrics, and their combinations. To evaluate the proposed model, they used a 10-fold cross validation, and repeated the process 100 times for each prediction model on each different time split to validate the prediction performance of MIMs. Results showed that they altogether improve defect classification and regression accuracy.

Ramay et al. [18] propose a deep neural network-based automatic approach that aims to predict the severity of bug reports. The proposed approach involves : 1) applying NLP techniques for text preprocessing of bug reports, 2) computing and assigning the emotion score for each of the processed bug report, 3) creating a vector for each preprocessed bug report, and finally 4) passing the vectors and the emotion scores to the deep neural network for severity prediction. We also evaluate the proposed approach on the history-data of bug reports. The evaluation process showed that the proposed approach outperforms the previous approaches by improving the f-measure by 7.90%.

Fan and others [19] proposed a deep learning-based strategy called DP-ARNN, as a guide to programming testing and code analysis, to predict potential code defects in programming. DP-ARNN used RNN to naturally create syntactic and semantic highlights from source code. The authors applied the proposed system to catch significant features that help to predict defects accurately. They used seven open-source projects to validate their method. Results showed that DP-ARNN outperforms the previous approaches and improves the F1-measure by 14% and AUC by 7%.

Mani and others [20] propose a bug report representation using a deep bidirectional RNN called DBRNN-A that discovers the syntactic and semantic features from long word sequences in an unsupervised manner. They use unfixed bug reports for training (about 70% of bugs from a open-source bug tracking system) which were ignored in previous studies. The other major contribution of this work is the delivery of a public dataset of bug reports from three open-source bug tracking systems (Google Chromium, Mozilla Core, and Mozilla Firefox). Results showed that DBRNN-A provides a higher rank-10 average accuracy compared to that other approached.

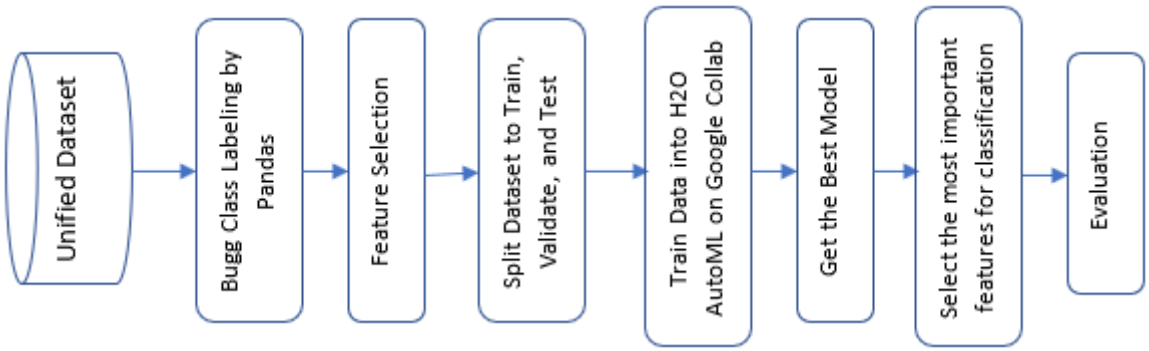
Bani-Salameh et al. [21] present a model that focuses on predicting priority level for each bug report. This work use a dataset built using bug reports that are taken from closed-source projects stored in the JIRA bug tracking system, which are used then to train and test the model. Also, they built a tool to help assign a priority level for the bug report automatically. Expermints involved applying RNN-LSTM neural network and comparing the results with other neural networks (SVM and KNN). Results found that the proposed model is 90% accurate in comparison with KNN (74%) and SVM (87%), and that RNN-LSTM improves the F-measure by 3% compared to SVM and 15.2% compared to KNN. Also, it

showed that LSTM reported the best performance results based on all performance measures (Accuracy = 0.908, AUC = 0.95, F-measure = 0.892).

Proposed Approach

Given a UNIFIED dataset of bugs from open-source software projects, this study uses H2O AUTOML [11] to detect and categorize code bugs. The process of assigning the category for code bugs consists of multi phases. Involves data collection, formatting priority of the bugs, feature selection, dataset training, applying H2O algorithm, and finally evaluation process (see Figure 1).

Figure 1. *Proposed framework flow*



.

Data Collection

The “unified bug dataset" had been used. This dataset produced from the Department of Software Engineering, the University of Szeged on December 20, 2019, and includes java classes with there are metrics and code Bugs extracted from 5 public bug datasets (PROMIS, Eclipse Bug, Bug Prediction, Bug catchers Bug, and GitHub Bug). The corresponding source code for each system in the datasets has been downloaded and then source code analysis has been performed using (OpenStaticAnalyzer-1.0-Metrics) to obtain a common set of source code metrics, then the result has been merged into one large dataset that consists of 47,618 elements with 72 feature. Therefore, this dataset is a good source of data for bug prediction models.

Labeling Bugs Categories

The bug category was labeled using the pd.cut function from the PANDAS library. The dataset was grouped into four classes (No, Low, Medium, Haigh) according to the number of bugs in each class, and the problem was converted to a classification problem.

Feature Selection

The dataset contains a large set of features that are still difficult to be processed us-ing the proposed algorithms. Thus, we choose a set of relevant features as an input for the proposed model. The unnecessary columns such as ('ID','Type','Name','LongName','Parent', 'Component', 'Path', 'Line', 'Column', 'EndLine', 'EndColumn') were dropped, while the features' selection process has been done automatically by H2O AUTOML.

Evaluation Metrics

The performance and efficiency of the classification algorithms were assessed using known metrics such as MSE, RMSE, LOGLOSS, and MEAN PER CLASS ERROR. Below is a description of the metrics used.

- *MES*: the MSE metric measures the root mean square of error or deviation. MSE squares the distance from the point to the regression line (these distances are called "errors") to eliminate the negative sign. The lower the MSE, the best model performance (see Equation 1).

$$\frac{1}{n}\sum_{i=1|}^{n}(x_i - y_i)^2 \qquad (1)$$

- *RMSE (Root Mean Squared Error)*: the RMSE metric evaluates how properly a version can are expecting a continuous value. The RMSE devices are like the anticipated target, that's beneficial for information if the dimension of the mistake is of challenge or not. The smaller the RMSE, the higher the model's performance.

- *Logloss:* the log loss rate can be used to evaluate the performance of binomial or polynomial classifiers. Different from the classification effect of the "Area Under Curve" test model on binary targets, the log loss estimates the closeness between the predicted value of the model and the actual target value.

H2O AUTOML Algorithms

H2O AutoML was used, which is a machine-learning algorithm for tabular data, which is part of the H2O machine-learning platform. H2O is easy to use and extensible and has a very active and participating user base in the open-source machine learning community. H2O AutoML can deal with lost or classified data, including a comprehensive modeling strategy for powerful functional combination components and the ability to easily deploy and use H2O models in enterprise production environments. Use Python to implement H2O AUTOML in a unified data set.

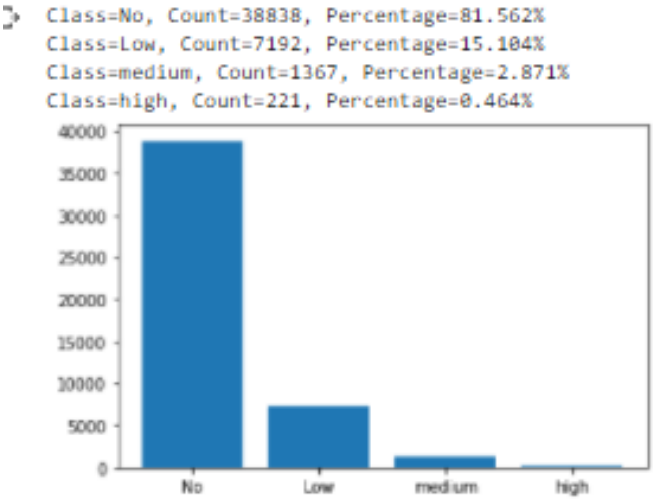
- *Input Variables:* Considering many factors (indicators), input variables were selected to predict category. Operators are columns reserved after features selection in the data set, except for the "bug" category.
- *Output variables:* In this paper, Bug class is the output variable used in the ML algorithms to be predicted.

Experiment and Results

The modified unified data set was divided into three parts (train, validation, test), the train part was used to train the models, and the verification part was used to set the hyperparameters and choose the best model for the problem, and the test part was used to verify Model performance in non-visual data.

Figure 2 shows the distribution of classes in the dataset during the data exploring process, which showed imbalanced classes, a problem in which the distribution of examples be-tween known classes is biased or skewed. The distribution range can range from slight offset to severe imbalance, with one example in the minority class for hundreds, thousands in the majority class or classes.

Figure 2. *The Distribution of the Classes during the Data exploring Process*



Therefore, the train part of data had been sampled using “SMOTE" (Synthetic Minority Oversampling Technology), an oversampling technology that uses the k algorithm to generate synthetic data for the nearest neighbors. SMOTE first selects random data from the minority class and then determines the k nearest neighbors of the data. Then the data placed between the random data and the randomly selected nearest neighbor k. the process Repeated until the minority category has the same proportion as the majority category. [22]

After that, the dataset had been trained into google Colab by the H2O Autml library. The H2O AutoML interface is designed to contain as few parameters as

possible. Customers only need to refer to their data set, define a response column, and optionally specify a time limit or the total number of trained models. Some of parameter that could be changed such as Max models, Nfolds, Max run-time secs, Balance classes, Stopping metric, Stopping rounds, Stopping tolerance, and Keep cross validation models [11]. Then, the model had been trained into the oversampled dataset, and the problem determined automatically by H20 as a multiclass.

After completing the training process, the result showed that 19 models belong to 6 families had been built during training (XGBoost, GLM , DRF , GBM , DeepLearning, DRF) A classification table is created every time AutoML is started. According to the nature of the problem, use standard metrics (the second column in the classification table) to classify the model. AUC is a metric used for binary classification problems, and the average error of each category is used for classification problems involving multiple categories. The standard ranking indicator for regression problems is variance. The user can choose to change the default ranking index of the ranking table. The result shows that the best 10 models for our dataset as presented in Table 1.

In addition, the result showed that the best model that solves the problem is "GBM 2 AutoML 20210415 132614", from the Gradient Boosting Machine family. which is a forward learning ensemble method that sequentially builds regression trees on all the features of the dataset in a fully distributed way each tree is built in parallel. This family is used for both Regression and Classification problems. The H2o suggests adapting the model settings as in Table 2 to give the best result in the unified dataset.

Table 1. *Best 10 models fitted dataset*

Model_Id	MPCR	LogLoss	RMSE	MSE	Algorithm
GBM_1_AutoML_20210524_001830	0.457483	0.941693	0.599223	0.359069	GBM
GBM_2_AutoML_20210524_001830	0.474377	0.921466	0.591464	0.34983	GBM
XGBoost_3_AutoML_20210524_001830	0.474787	0.727399	0.5075	0.257556	XGBoost
GBM_3_AutoML_20210524_001830	0.477752	0.981109	0.616594	0.380189	GBM
GBM_4_AutoML_20210524_001830	0.479495	1.00402	0.62688	0.392978	GBM
GBM_grid__1_AutoML_20210524_001830_model_2	0.486995	0.868863	0.566413	0.320824	GBM
GBM_grid__1_AutoML_20210524_001830_model_1	0.513725	0.573223	0.43698	0.190951	GBM
GBM_5_AutoML_20210524_001830	0.525829	0.992099	0.622708	0.387766	GBM
XGBoost_1_AutoML_20210524_001830	0.528332	0.776078	0.529233	0.280087	XGBoost
GBM_grid__1_AutoML_20210524_001830_model_3	0.545588	1.21901	0.702968	0.494164	GBM

Table 2. *Models' parameter summary*

id	parameter	value
1	number_of_trees	7
2	number_of_internal_trees	28
3	model_size_in_bytes	23704.0
4	min_depth	6
5	max_depth	6
6	mean_depth	6
7	min_leaves	54

8	max_leaves	64
9	mean_leaves	62.857143

The METRICS of the best model into train, validate, and test parts of unified dataset are mentioned in Table 3. Figure 3 shows a summary of the cross-validation metrics for the best model.

Table 3. *The metrics of the best model*

Metric	train_data	cross_validation	trest_data
MSE	0.37648429685384704	0.3775879368430466	0.35906867464119113
RMSE	0.6135831621335832	0.614481844193176	0.5992233929355488
LogLoss	0.9703039488095602	0.9732992146307755	0.9416925324811019
Mean Per-Class Error	0.29158092018227255	0.2918014111421432	0.4574829531658866

Figure 3. *A summary of the cross-validation metrics*

Cross-Validation Metrics Summary:

		mean	sd	cv_1_valid	cv_2_valid
0	accuracy	0.7300272	0.0053789173	0.7262237	0.7338307
1	auc	NaN	0.0	NaN	NaN
2	aucpr	NaN	0.0	NaN	NaN
3	err	0.2699728	0.0053789173	0.27377626	0.26616934
4	err_count	14693.0	292.74222	14900.0	14486.0
5	logloss	1.0175679	6.08243E-4	1.0179979	1.0171378
6	max_per_class_error	0.42002004	0.013946684	0.41015828	0.42988184
7	mean_per_class_accuracy	0.7300389	0.0054462566	0.72618777	0.73388994
8	mean_per_class_error	0.26996115	0.0054462566	0.27381223	0.26611006
9	mse	0.39997348	1.5427692E-4	0.40008256	0.39986438
10	pr_auc	NaN	0.0	NaN	NaN
11	r2	0.68002087	1.5002872E-4	0.6799148	0.68012697
12	rmse	0.63243455	1.21970654E-4	0.6325208	0.6323483

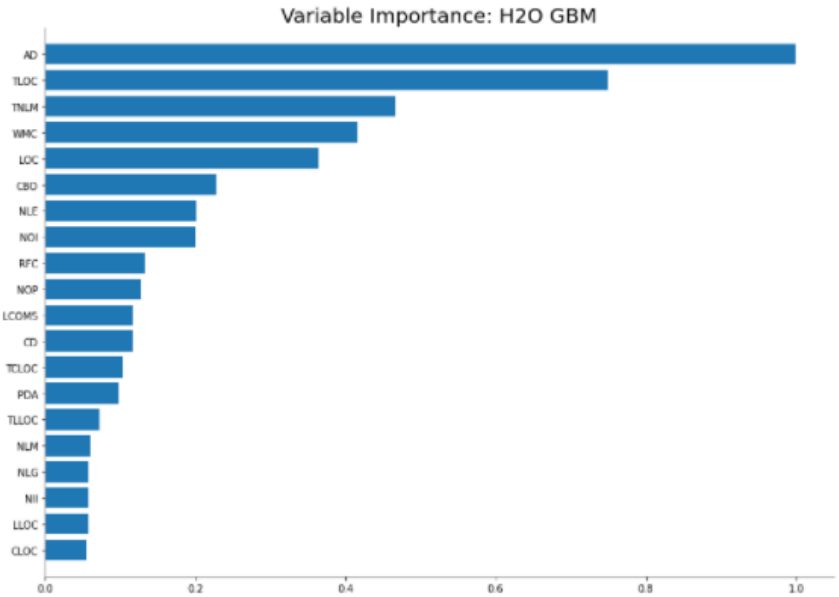
Moreover, the result showed that the most 20 importance features that effect classification for the best model were as mentioned in Figure 4.

Figure 4. *The most 20 important features*

CLASS\METRIC	WMC	CBO	RFC	MCABE	LCOM
InGameController	132	84	593	6.838	0.426
InGameController	120	83	573	5.235	0.032
CHANGE	-12	-1	-20	-1.603	-0.394
SimpleCombatModel	14	29	120	9.923	0.869
SimpleCombatModel	12	25	94	7.273	0.867
CHANGE	-2	-4	-26	-2.65	-0.002
Monarch	33	18	123	3.586	0.8
Monarch	32	18	119	3.39	0.797
CHANGE	-1	0	-4	-0.196	-0.003
ColonyPlan	37	43	229	9.375	0.869
ColonyPlan	36	43	226	9.452	0.867
CHANGE	-1	0	-3	0.077	-0.002
SimpleMapGenerator	22	64	255	9.25	0.688
SimpleMapGenerator	17	49	196	8.933	0.677
CHANGE	-5	-15	-59	-0.317	-0.011
TerrainGenerator	25	23	183	11.625	0.769
TerrainGenerator	20	22	166	10.317	0.767
CHANGE	-5	-1	-17	-1.308	-0.002

The statistical significance of each variable in the data in terms of its effect on the model was represented by variable importance. The variables were listed in descending order of importance. The percentage values represent the proportion of importance across all variables, scaled to 100%. The algorithm determines how to compute the importance of each variable (see Figure 5).

Figure 5. *The importance of each variable*



Conclusion and Future Works

All in all, this research provides a framework that can automatically assign appropriate classes to code bugs, thereby avoiding time-consuming and resource-limited software testing. The proposed structure involves using the library H2O AUTOML to determine the best model that matches the bugs unified dataset .the results showed that the best model is “GBM 2 AutoML 20210415 132614" from the Gradient Boosting Machine family with an accuracy of 73% and mean per class error of 46%. In addition, Features that most affect bug classification have been identified.

The ability to use H2O AutoML to classify bugs in Unified dataset has been demonstrated, H2O exceeds expectations in terms of ease of use and scalability, active customer base, and open-source machine learning community.

In future work, we plan to perform the following operations: perform the same experiment on new datasets, metrics, and other types of bugs. Use different types of AutoML libraries such as AutoKeras and Autogloun to train new models and compare the results between them. development of a new tool based on the best model that fits the data set for automatic bug classification and correction.

References

[1] S. M. Abozeed, M. Y. ElNainay, S. A. Fouad, and M. S. Abougabal, “Software bug prediction employing feature selection and deep learning,” in 2019 Inter-national Conference on Advances in the Emerging Computing Technologies (AECT). IEEE, 2020, pp. 1–6.

[2] R. Malhotra, “A systematic review of machine learning techniques for software fault prediction,” Applied Soft Computing, vol. 27, pp. 504–518, 2015.

[3] Z. Tóth, “New datasets for bug prediction and a method for measuring maintainability of legacy software systems,” Ph.D. dissertation, University of Szeged, 2019.

[4] A. Perera, “Using defect prediction to improve the bug detection capability of search-based software testing,” in 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2020, pp. 1170–1174.

[5] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy, “A public unified bug dataset for java and its assessment regarding metrics and bug prediction,” Software Quality Journal, pp. 1–60, 2020.

[6] R. Ferenc, D. Bán, T. Grósz, and T. Gyimóthy, “Deep learning in static, metric-based bug prediction,” Array, vol. 6, p. 100021, 2020.

[7] E. Kaen and A. Algarni, “Feature selection approach for improving the accuracy of software bug prediction.”

[8] F. Khan, S. Kanwal, S. Alamri, and B. Mumtaz, “Hyper-parameter optimization of classifiers, using an artificial immune network and its application to software bug prediction,” IEEE Access, vol. 8, pp. 20 954–20 964, 2020.

[9] R. S. Wahono, “A systematic literature review of software defect prediction,” Journal of Software Engineering, vol. 1, no. 1, pp. 1–16, 2015.

[10] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, “Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques,” Expert Systems with Applications, vol. 144, p. 113085, 2020.

[11] A. Candel, V. Parmar, E. LeDell, and A. Arora, “Deep learning with h2o,” H2O. ai

Inc, 2016.
[12] M. Brumfield, "A deep learning approach to predict software bugs using micro patterns and software metrics," Ph.D. dissertation, Mississippi State University, 2020.
[13] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, "Vulnerability prediction from source code using machine learning," IEEE Access, vol. 8, pp. 150 672–150 684, 2020.
[14] E. A. Felix and S. P. Lee, "Predicting the number of defects in a new software version," PloS one, vol. 15, no. 3, p. e0229131, 2020.
[15] A. Trautsch, S. Herbold, and J. Grabowski, "Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction," in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2020, pp. 127–138.
[16] B. Wójcicki and R. Dabrowski, "Applying machine learning to software fault prediction," e-Informatica Software Engineering Journal, vol. 12, no. 1, 2018.
[17] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, pp. 311–321.
[18] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu, and I. Illahi, "Deep neural network-based severity prediction of bug reports," IEEE Access, vol. 7, pp. 46 846–46 857, 2019.
[19] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," Scientific Programming, vol. 2019, 2019.
[20] S. Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effective-ness of deep learning for bug triaging," in Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, 2019, pp. 171– 179.
[21] H. Bani-Salameh, M. Sallam et al., "A deep-learning-based bug priority prediction using rnn-lstm neural networks," e-Informatica Software Engineering Journal, vol. 15, no. 1, 2021.
[22] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: syn-thetic minority over-sampling technique," Journal of artificial intelligence research, vol. 16, pp. 321–357, 2002.