# ML-powered Multi-Bug Detection

of Python Code for Python Developers by Python Modules

*An increase in the number of software projects being written and put into production over the last two decades specifically has made the issue of bug detection an all the more costly problem to solve. Owing to the popularity of Python and the desire from developers to have an all-in-one solution for bug detection, this paper focuses on establishing the best Machine Learning (ML) method that can be applied to classify 10 different types of bugs from non-buggy code in Python. As there exists no such dataset (of labeled, buggy code) readily available in Python, this paper proposes a data-processing tool that artificially injects bugs into non-buggy code and labels the data accordingly. Using Scikit-learn and Tensorflow as the primary modules to vectorize the processed data, train ML models on the data, and eventually validate and test the accuracy of these models, this paper compares the performance of single classifier methods (such as LinearSVC, Multinomial Naive Bayes, Decision Trees, Neural Networks, etc.) to ensemble classification methods (such as Random Forests, AdaBoost, etc.). Having established Neural Networks and GradientBoosting Classifiers to be the best single and ensemble classifiers respectively, this paper furthers the optimization process by using Keras' Autotuner to search the hyperparameter space for the combination that maximizes classification accuracy of both methods on the validation data.*

*Using over 83,000 lines of labeled code to train and validate the models and over 54,000 lines of labeled code to test the performance of the best classifiers, it was found that Tensorflow's Neural Networks have the best classification accuracy with 75%. While it was expected that the ensemble methods might do better when solving this problem - especially given the need to classify a variety of bugs thereby making the decision boundary more convoluted, it was found that the classification accuracy of most of the baseline ensemble models was just over 20%. The ExtraTreesClassifier and GradientBoostingClassifier showed the most promise in preliminary trials however despite optimizing their hyperparameters, these classifiers were still not able to achieve classification accuracy substantially over 20%. The considerable performance gap between Neural Networks and Ensemble Classifiers in this paper could be attributed to either 1) Neural Networks' ability to gain a deeper understanding of code features and what therefore what qualifies as a bug or 2) poor vectorization using scikit-learn's TFIDF vectorizer or both.*

## 1) Introduction

Information technology (IT) spending on enterprise software worldwide has more than doubled from 2009 to 2022 [1]; that is just one metric of many that show the increase in both development and usage of software projects recently - specifically in the last decade. Moreover, the proportion of budget allocated to quality assurance and testing as a percentage of IT spend underwent a substantial increase from 18% in 2012 to 40% in 2019 [2]. This increase is well explained; an increase in the amount of code written by developers prone to some level of error will lead to an increase in the number of bugs within software that is now being used by a larger audience. Besides, studies estimate that the cost of fixing a bug in code increases 100 fold on average as the project goes from development to production [3].

'Forbes, in their research and analysis of 'technical skills with huge demand' list, has put Python seventh on the list, with a 456% growth rate last year. According to the Stack Overflow website, the questions with a Python tag have the maximum number of views in 2018 with about 10.5% of overall question views each month.' [4] Despite the increasing popularity in Python making it one of the primary development languages in the world currently, academic research in the field of ML-based bug detection has not been focused on this language or resolving the bugs thereof. A survey of software developers - most with varying levels of professional experience in producing code - ranked detection of a wider variety of bugs as their most desired feature in a bug detector as can be seen from the chart below.

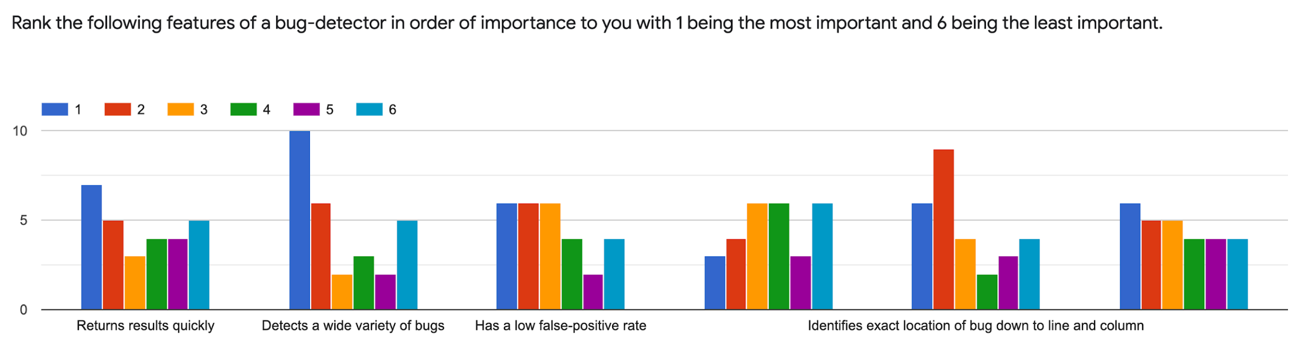


*Figure 1.1) Most desired features in a bug detector according to a survey of software developers*

Yet surprisingly, current state-of-the-art research tools in this field have targeted detecting individual bug types in their respective languages instead of an all-in-one approach. Given the above information, this paper has chosen to focus on solving a predominant, high-cost problem in the field of software development using ML approaches to address the current gaps in academic research in this field.

## 1.1) Novelty

Given that extensive research has already been conducted in this field with promising results, this paper seeks to add value to the proposed user community of Python software developers by proposing the following novel ideas - which have not received substantial attention - especially in the format below:

1. This paper will attempt to compare and contrast the performance of multiple ML classifiers (both single and ensemble methods) in identifying a variety of bug types including but not limited to argument swapping, missing function calls, absence of parenthesis, mismatched indentation, and incorrect operators.
2. This paper will introduce a data-processing tool that allows for bugs to be artificially induced into Python code while extracting the same from project folders thereby creating a neatly labeled dataset of buggy and non-buggy Python code. Not only does this solve the absence of such a dataset presently existing for Python (by creating a new publicly available one for the training and testing process), but it will allow future users to create their own datasets based on the types of bugs that they would like to test for in the future.
3. Current academic research in this field is clearly divided into two approaches: single methods and ensemble methods. The difference between the two may be important to researchers but to an end-user, classification accuracy is paramount - no matter the performance of the tool relative to other similar approaches. In light of this, this paper not only breaks the boundary by comparing a wide variety of both single and ensemble classifier methods against each other but also applies Keras' Autotuner to refine the hyperparameters of the best of these methods. This results in the end-user not only being exposed to a much wider collation of results than has been previously available but also benefiting from the repeated levels of manual and automated selection and optimization of models.

## 1.2) Research Questions to be Answered

The proposed novelty of the work done in this paper can result in a very broad scope when attempting to analyze the results achieved. For this reason, the following research questions will be considered key throughout the development phase and when attempting to ascertain the final contributions and results of the work done:

1. How well do ensemble methods from Scikit-learn do when attempting to classify non-buggy data vs. buggy data in Python code when a) multiple bug types need to be detected and classified and b) the bugs are artificially introduced?
2. How well do neural networks from Tensorflow do when attempting to classify non-buggy data vs. buggy data in Python code when a) multiple bug types need to be detected and classified and b) the bugs are artificially introduced?
3. Given the problem being solved, which ML method performs the best and how do they compare against each other? Can Keras' Autotuner substantially boost the performance of the best methods?
4. Is the performance of the best classifier comparable to industry and research standards? Are there any takeaways from the work done that could be applied to future research in the field?

# 2) Methods

Owing to the diversity of work being done in this paper, it was all the more important to clearly define the experimental setup and methods, their implementation, usage, and final application.

## 2.1) Setup

The following steps were taken to set up the entire experiment from ingesting non-buggy Python code from project folders to setting up the Keras Autotuner to optimize model hyperparameters:

1. Bug-free Python projects were downloaded from popular, well-reviewed Python repositories by different authors on Github containing Python code written for a multitude of purposes. eg: OmkarPathak/Python-Programs: My collection of Python Programs. This ensured that the code extracted did not follow a particular coding style and contained substantial diversity in terms of functionality and function.
2. The codebases selected were divided into a training dataset and a test dataset. The training dataset was further subdivided into training (80% of the code) and validation (20%) of the code - ensuring that each of the three categories for training and testing an ML algorithm was covered.
3. As each line of code was extracted, each of the following ten bugs was attempted to be induced into the line artificially if possible: Arguments Swapped, Extra Argument, Extra Indentation, Incorrect Comparison, Less Arguments, Less Indentation, Missing Function Call, Missing Outermost Parentheses, Off by One and Variable Referenced before Assignment. The buggy lines (wherever possible), as well as the original non-buggy line, were written to a CSV file with the corresponding label.
4. The CSV file was read into a Pandas dataframe and the labels were converted to numeric values for easier classification. The choice of Pandas was made since a vast majority of the methods being used were from the Scikit-learn module for which Pandas data frames provide easy integration.
5. An 80: 20 (training: validation) split was created in the case of the training data. The lines of code were vectorized by the TFIDF vectorizer provided by Scikit-learn's text extraction module.
6. The following ML algorithms from Scikit-learn were then applied to the training data with the objective being to maximize classification accuracy on the training data: LinearSVC, MultinomialNB, LogisticRegression, DecisionTreeClassifier, KNeighborsClassifier, RandomForestClassifier, AdaBoostClassifier, ExtraTreesClassifier, GradientBoostingClassifier, BaggingClassifier, and VotingClassifier. The trained models were compared against each other with regards to classification accuracy on the validation dataset as well as using a 5-fold cross-validation.
7. A sequential Neural Network model consisting of a combination of embedding, dense, convolutional, and pooling layers was also trained on the model.
8. The best models from above were reconstructed as a method taking in a wide variety of hyperparameters as the input. This allowed for the models to then be optimized for validation accuracy using Keras' AutoTuner.

## 2.2) Challenges Addressed

As with any engineering problem (and especially true of ML owing to its at times black-box nature), multiple challenges and problems arose during the development, setup, and execution phases of the work proposed. The key challenges among these were addressed in the following ways:

1. Given the wide variety of code being ingested and then processed to contain artificially-induced bugs, the first problem faced was ensuring that
    a. In-line comments or block comments were ignored successfully
    b. Bugs were introduced in the correct places, eg: argument swapping bugs must not be introduced in function definitions but only in function calls
    c. Bugs that were introduced at a specific point in the line of code, did not affect the rest of the line - which would otherwise harm classification
    d. Any type of Python line could be extracted, processed, and dealt with accordingly
    e. Bugs of each type were introduced at all possible locations

    The above issues were solved by ensuring that each of the functions generating bugs in extracted code was robust on a variety of edge cases thereby ensuring desired performance. Furthermore, it was ensured that each extracted line was a valid, correctly-formatted line of code before adding it to the training dataset or artificially generating bugs in said line.
2. Using the ‘ast’ module proved to make vectorization a more complicated problem - which if not solved correctly would adversely affect classification accuracy. The original attempt for classification involved passing in two inputs to the classifier: 1) the line of code being examined and 2) the prior lines of code within the same function. This was proving to be a complicated problem to solve especially given the perceived gain in accuracy at this stage of building the tool. While the code is still currently set up to extract the prior lines of code within a function up until the line being examined, the scope of the problem was simplified by resorting to using only the current line of code as input and Scikit-learn’s TFIDFVectorization or Tensorflow’s TextVectorization as the vectorizer.
3. The original input pipeline involving the usage of a CSV file to store the line and its label was proving to be slow and inefficient when training the Neural Network - and was likely harming accuracy as well as can be seen from the poor performance in the graphs below.

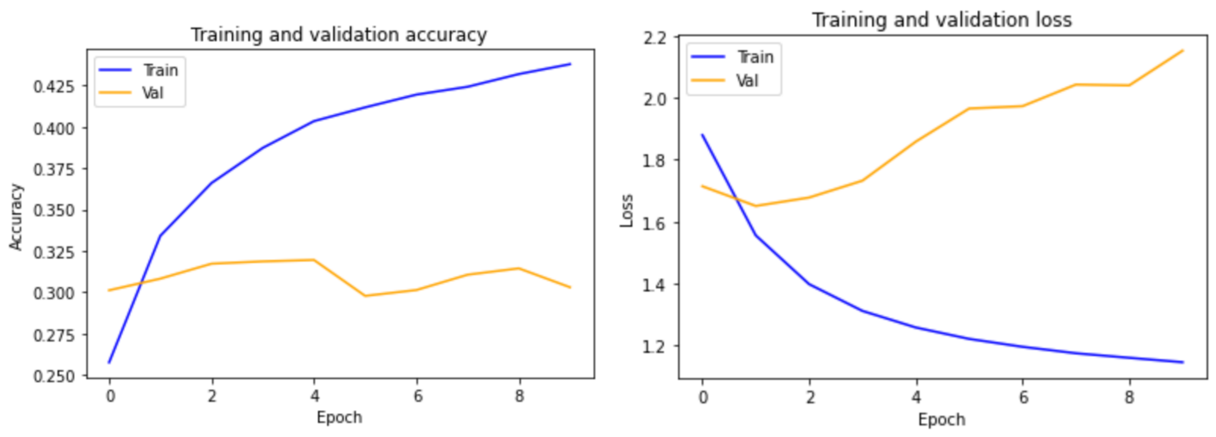


*Figure 2.2.1) Training and Validation Loss and Accuracy when using first Input Pipeline with basic Tensorflow Neural Network*

To this end, a second input pipeline was added to the tool that involves the usage of Tensorflow’s preprocessing.text_dataset_from_directory(). The pipeline now saves each extracted and processed line to a .txt file that contains all the lines from the same file. However, based on the type of line - ‘No bug’, ‘Arguments swapped’, ‘Extra Argument’, etc., the line is added to .txt file with the same name but in a folder corresponding to its label. The folder structure can be found in Figure 2.2.2.

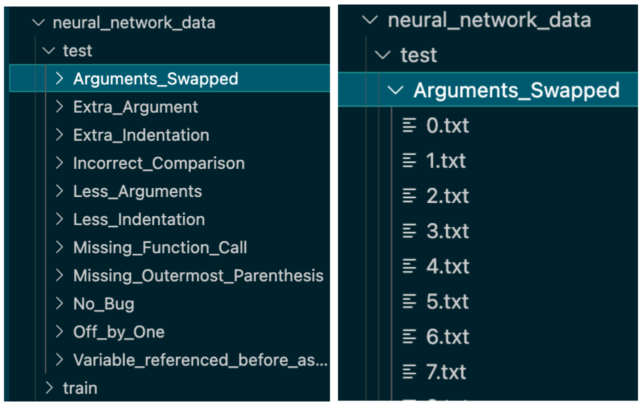


*Figure 2.2.2) Folder Structure of Formatted Input Data for Tensorflow*

This input pipeline now allows users to train/test the tool on pre-formatted data following two different structures thereby increasing the versatility of the tool. More importantly, utilizing this input pipeline allowed for leverage of other preprocessing tools such as caching and prefetching the data. The gain in epoch speed allowed by these resulted in substantially faster turn-around times when using Keras' Autotuner to test multiple neural networks on a multitude of epochs.

## 3) Results

This section covers the major results of the work proposed in the previous sections. A brief explanation of each graph/figure/result may be found in accompaniment of the same however any detailed discussion will be found in Section 4 when answering the research questions proposed in Section 1 and comparing this work to work from external research.

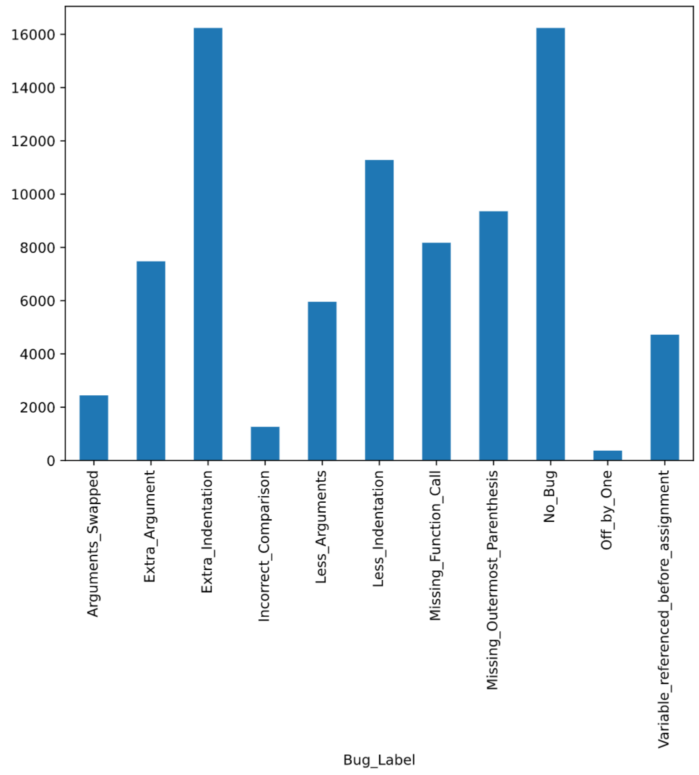


*Figure 3.1) Lines of Code per Bug Type in Training Data*

## 3.1) Single Classifier Methods

Before moving to ensemble classifiers, it was necessary to understand the performance of single classifiers in solving the problem of multi-bug classification. This information would help us understand which single classifiers we might want to build ensemble classifiers out of. To this end, multiple models from Scikit-learn were compared against each other in a 5-fold cross-validation format. The results of the same can be seen in Figure 3.1.1.

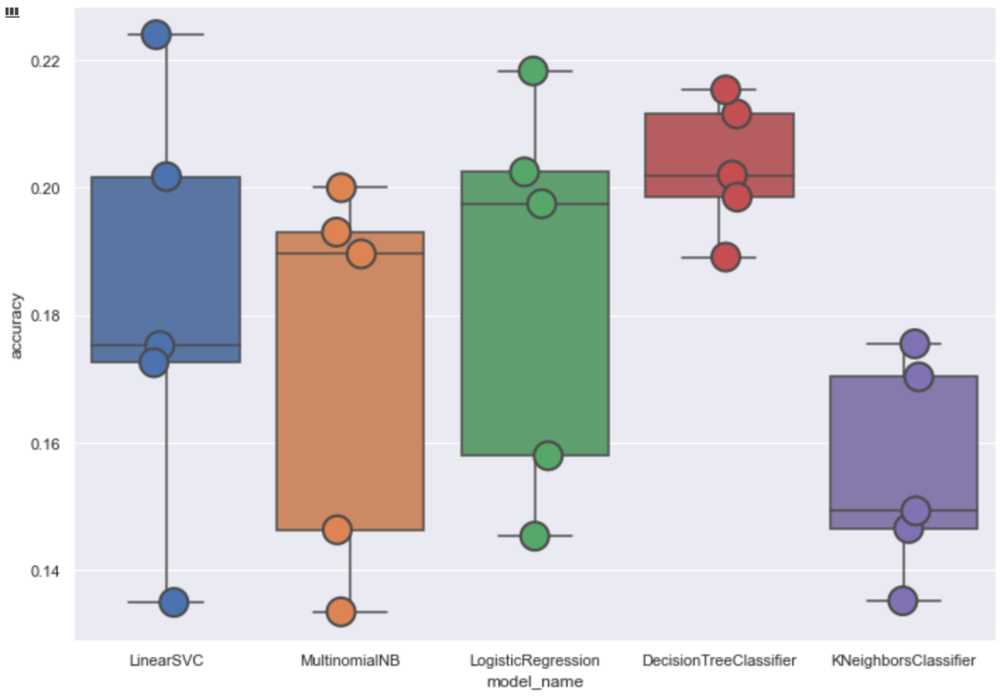


*Figure 3.1.1) 5-Fold Cross-Validation Accuracies of Single Classifiers*

While the highest point of the classification accuracy of LinearSVC is the highest, it is the DecisionTreeClassifier that performs the best on average as can be seen from the above graph and on the validation data as can be seen from the classification reports in Appendix Section 7.2. This is encouraging as a vast majority of the Ensemble Classifier methods use this classifier as the base estimator.

## 3.2) Ensemble Methods

Understanding the performance of ensemble classifiers in solving the problem of multi-bug classification required more analysis and training than did the same for single classifiers. This is largely due to the higher number of hyperparameters to tune in ensemble models as well as the increased time taken to train and evaluate these models - making it harder to repeatedly iterate on them. Fortunately, from the results presented in Section 3.1, it was confirmed that the native base estimator used by all the ensemble methods that Scikit-learn has to offer - the DecisionTreeClassifier was most well suited to solve this problem anyway.

Before all of the ensemble classifiers were trained and evaluated, the performance of two of the most basic ensemble classifiers: AdaBoost and RandomForests were compared with the performance of LinearSVC. It must be noted that comparing the performance of RandomForests to DecisionTreeClassifiers would be redundant since the former comprises of the latter. As can be seen from Figure 3.2.1, the performance of RandomForestClassifiers when it came to 5-fold cross-validation was indeed substantially better than LinearSVC. Additionally, while LinearSVC's maximum accuracy was greater than AdaBoost, it must be noted that AdaBoost had a higher average and a lower range indicating more stable performance.

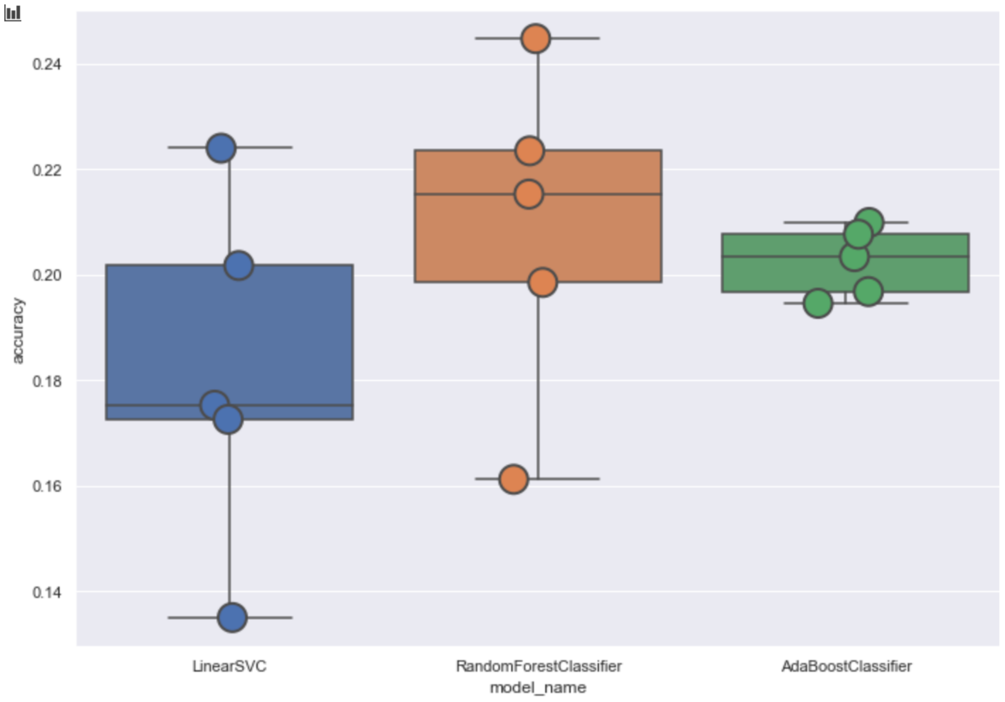


*Figure 3.2.1) 5-Fold Cross-Validation Accuracies of LinearSVC vs. Basic Ensemble Classifiers*

Despite the elevated performance of the ensemble methods above in comparison to the LinearSVC single classifier, the increase in classification accuracy for the ensemble methods above was not as high as expected. In view of this, four different classifiers that offer easy evaluation and comparison amongst themselves were fit on the training data - each time with an increase in the number of base estimators they comprised of. Their validation accuracy was then plotted in Figure 3.2.2. The goal of this was to understand if the lack of sufficient jump in accuracy described from Figure 3.2.1 could be attributed to sub-optimal models created by the default configuration of the above classifiers.

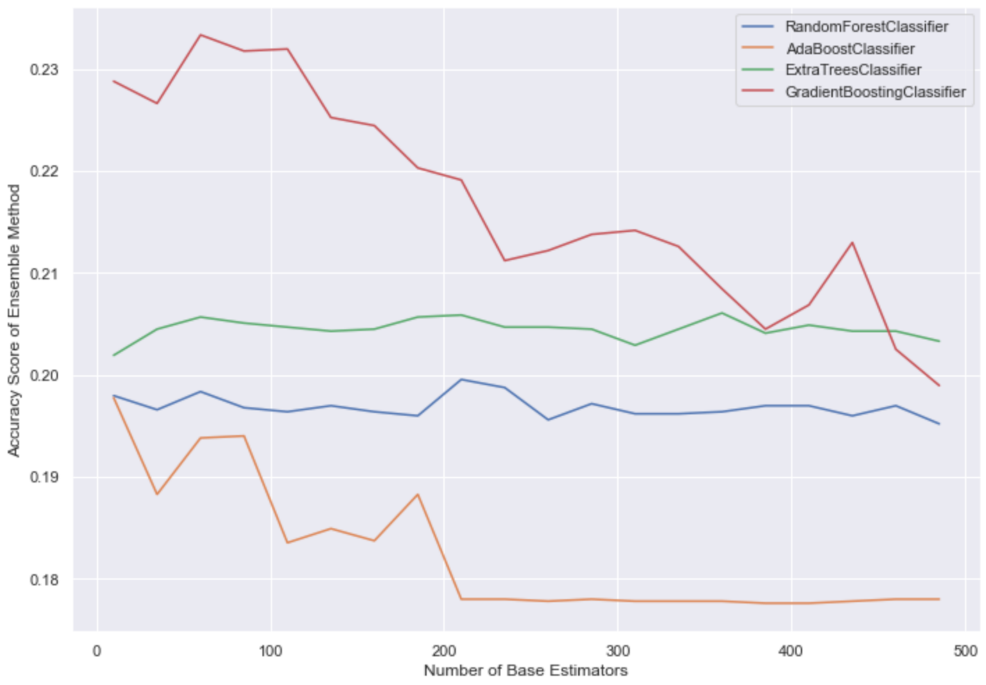


*Figure 3.2.2) Validation Accuracy of Ensemble Method vs. Number of Base Estimators*

From Figure 3.2.2, the best ensemble classifiers are clearly the GradientBoostingClassifier and the ExtraTreesClassifier. However it must be noted that the accuracy of these classifiers is still far below what was expected and that the accuracy of all of these classifiers tapers off when more than 100 base estimators are used. Having narrowed down the best classifiers from the above, the models were parametrized and fed to the Keras AutoTuner in an attempt to see if altering other combinations of hyperparameters alongside altering the number of base estimators from 0 - 100 would result in an improvement in accuracy. However, this was not the case as even the best model generated did not consistently achieve over 25% accuracy in classification on the validation dataset (as can be seen in the 'project_first_complete.ipynb' Jupyter Notebook in the deliverable repository. As a final comparison attempt on the dataset, Figure 3.2.3 was generated to at the very least have a comparative understanding of the performance of all the ensemble classifiers relative to each other.

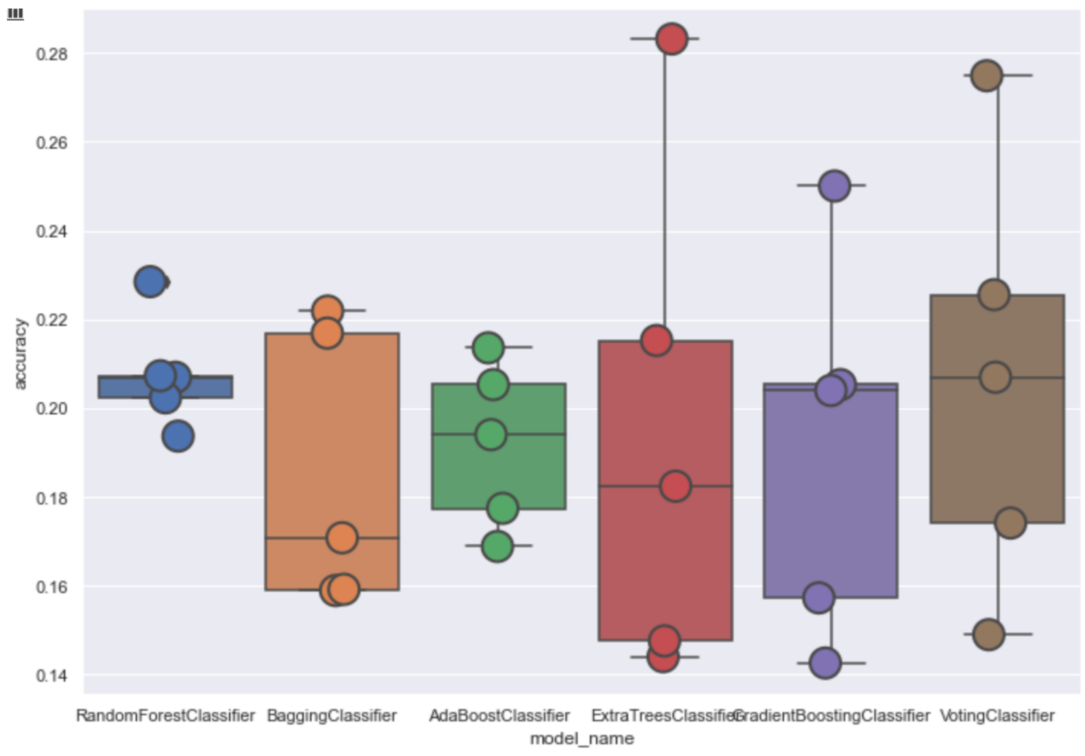


*Figure 3.2.3) 5-Fold Cross-Validation Accuracies of all Ensemble Classifiers*

## 3.3) Neural Networks

Given the superior performance of even a basic Neural Network model when using the Pandas dataframe input pipeline, the next obvious step was to further investigate the performance of Neural Networks with the Tensorflow optimized pipeline and vectorization to determine if that would further increase the classification accuracy before such models were further developed. This was done using a Binary TextVectorization layer and simple sequential model consisting of a singular dense layer. As can be seen from Figure 3.3.1, this attempt was successful at confirming the improved accuracy of Neural Networks with the new pipeline and was grounds for continued work in this direction.

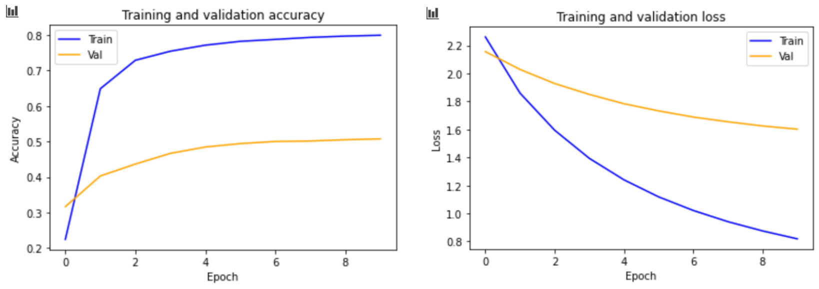
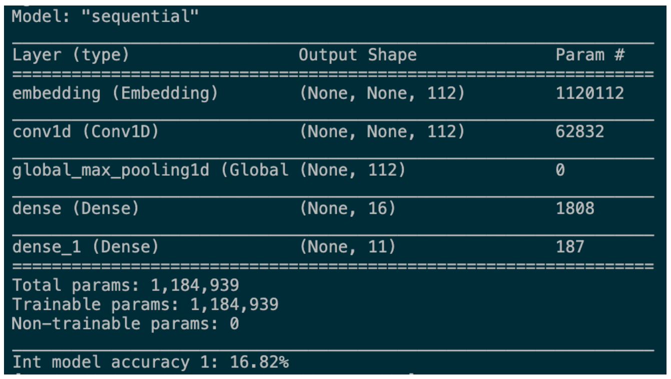
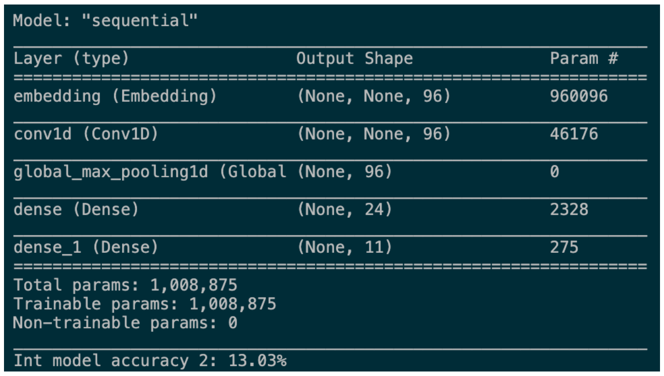


*Figure 3.3.1) Training and Validation Loss and Accuracy of Binary Neural Network Model*

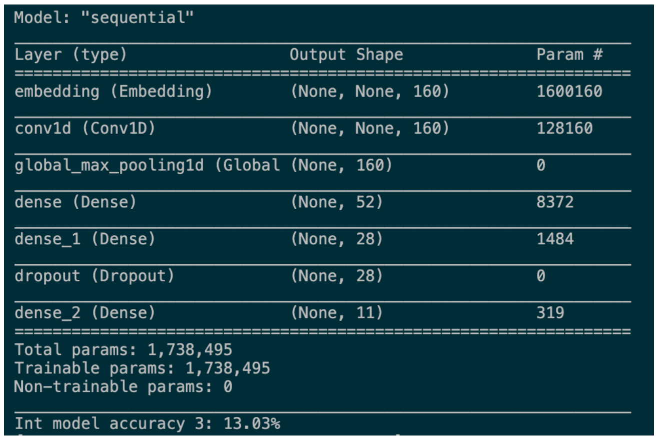
Following the results from the above, multiple combinations of sequential Neural Network models were created and run through Keras AutoTuner with the summary and results of the best four of them presented below.



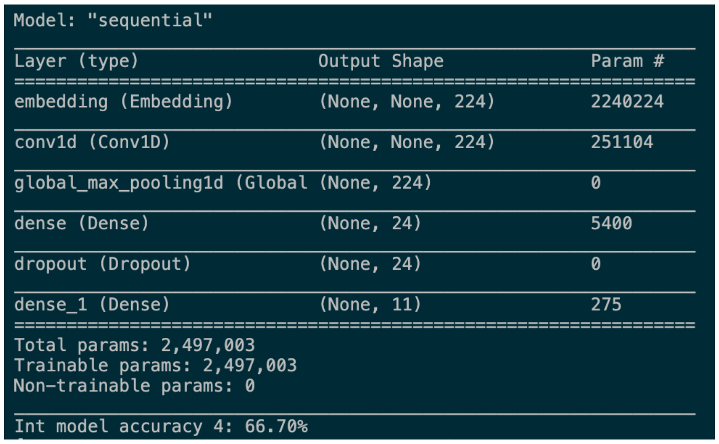
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 112)         1120112
_________________________________________________________________
conv1d (Conv1D)              (None, None, 112)         62832
_________________________________________________________________
global_max_pooling1d (Global (None, 112)               0
_________________________________________________________________
dense (Dense)                (None, 16)                1808
_________________________________________________________________
dense_1 (Dense)              (None, 11)                187
=================================================================
Total params: 1,184,939
Trainable params: 1,184,939
Non-trainable params: 0
_________________________________________________________________
Int model accuracy 1: 16.82%
```



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 96)          960096
_________________________________________________________________
conv1d (Conv1D)              (None, None, 96)          46176
_________________________________________________________________
global_max_pooling1d (Global (None, 96)                0
_________________________________________________________________
dense (Dense)                (None, 24)                2328
_________________________________________________________________
dense_1 (Dense)              (None, 11)                275
=================================================================
Total params: 1,008,875
Trainable params: 1,008,875
Non-trainable params: 0
_________________________________________________________________
Int model accuracy 2: 13.03%
```



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 160)         1600160
_________________________________________________________________
conv1d (Conv1D)              (None, None, 160)         128160
_________________________________________________________________
global_max_pooling1d (Global (None, 160)               0
_________________________________________________________________
dense (Dense)                (None, 52)                8372
_________________________________________________________________
dense_1 (Dense)              (None, 28)                1484
_________________________________________________________________
dropout (Dropout)            (None, 28)                0
_________________________________________________________________
dense_2 (Dense)              (None, 11)                319
=================================================================
Total params: 1,738,495
Trainable params: 1,738,495
Non-trainable params: 0
_________________________________________________________________
Int model accuracy 3: 13.03%
```



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 224)         2240224
_________________________________________________________________
conv1d (Conv1D)              (None, None, 224)         251104
_________________________________________________________________
global_max_pooling1d (Global (None, 224)               0
_________________________________________________________________
dense (Dense)                (None, 24)                5400
_________________________________________________________________
dropout (Dropout)            (None, 24)                0
_________________________________________________________________
dense_1 (Dense)              (None, 11)                275
=================================================================
Total params: 2,497,003
Trainable params: 2,497,003
Non-trainable params: 0
_________________________________________________________________
Int model accuracy 4: 66.70%
```

*Figure 3.3.2) Summary and Accuracy of 4 best Neural Networks from Keras AutoTuner (saved in deliverable repository)*

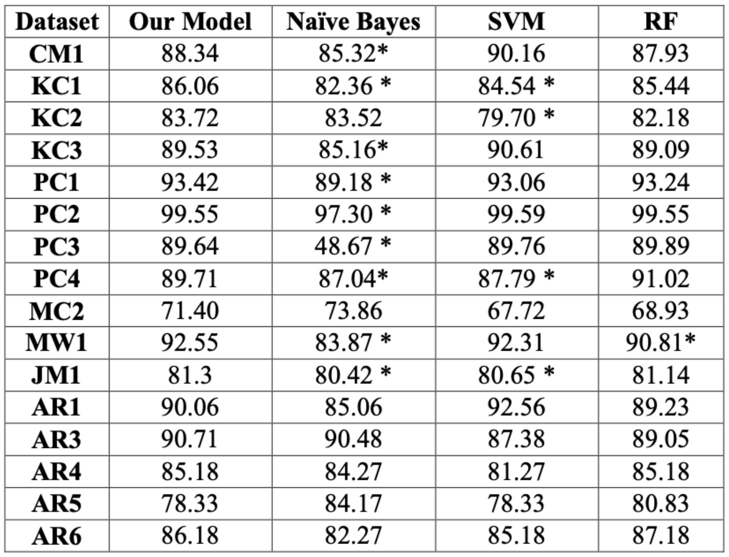
While not shown in Figure 3.3.2, each of the above four models achieved high levels of accuracy (between 71% and 75%) on the validation dataset - which was the benchmark against which the Keras AutoTuner was being run. However, their substantially decreased performance on the test dataset as can be seen from the figure above indicates that all except for the last model were suffering from overfitting since the model was being trained only on the training dataset but the hyperparameters were being pushed to optimize for the validation dataset - hence the decrease in performance when tested on entirely new data. The last model's combination of activation functions, number of neurons and dropout allowed it to avoid this problem.

# 4) Discussion

Given the extensive results presented in Section 3 alongside the explanation for each, this section will briefly conclude the results of the work done by comparing the work above first to external research done in the field - in so far as such a comparison is possible given the relative novelty and scope of this work. Finally, the research questions posed at the start of the paper will be answered succinctly, the results of which have already been presented above.

## 4.1) Comparison to External Research

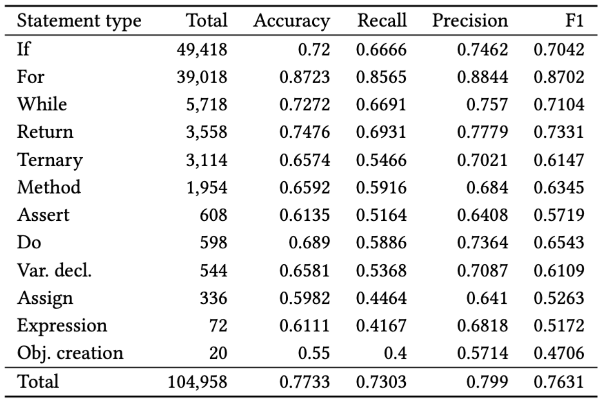
One of the key novel features of the work done in this paper was the variety of bugs attempting to be classified by the different methods evaluated. Given this, the only comparable papers currently in academic research that also use ML are all concerned with ensemble methods. The final classification results of one of these papers can be seen in the figure below with additional results available in Appendix Section 7.2.4. As can be seen from the figure below, there is much variability in the classification accuracy of ensemble methods on different buggy datasets. This indicates that the combination of a plethora of bugs in the dataset used in this paper coupled with weaker vectorization methods was likely a major contributor to the poorer performance of the ensemble methods in this paper. The work presented by Singh & Verma in 2018 however did not investigate either the number of ensemble models or optimization thereof that this work did. Hence, the belief is that future work done on this paper to improve the vectorization methods used will bring about comparable, if not better results than the ones seen below. This can be said since in the results presented in Section 3, despite the improved performance of RandomForests over single classifiers, work was done to show that the GradientBoostingClassifier and the ExtraTreesClassifier could be optimized t far surpass the performance of RandomForests using the Keras AutoTuner.



| Dataset | Our Model | Naïve Bayes | SVM | RF |
|---|---|---|---|---|
| **CM1** | 88.34 | 85.32* | 90.16 | 87.93 |
| **KC1** | 86.06 | 82.36 * | 84.54 * | 85.44 |
| **KC2** | 83.72 | 83.52 | 79.70 * | 82.18 |
| **KC3** | 89.53 | 85.16* | 90.61 | 89.09 |
| **PC1** | 93.42 | 89.18 * | 93.06 | 93.24 |
| **PC2** | 99.55 | 97.30 * | 99.59 | 99.55 |
| **PC3** | 89.64 | 48.67 * | 89.76 | 89.89 |
| **PC4** | 89.71 | 87.04* | 87.79 * | 91.02 |
| **MC2** | 71.40 | 73.86 | 67.72 | 68.93 |
| **MW1** | 92.55 | 83.87 * | 92.31 | 90.81* |
| **JM1** | 81.3 | 80.42 * | 80.65 * | 81.14 |
| **AR1** | 90.06 | 85.06 | 92.56 | 89.23 |
| **AR3** | 90.71 | 90.48 | 87.38 | 89.05 |
| **AR4** | 85.18 | 84.27 | 81.27 | 85.18 |
| **AR5** | 78.33 | 84.17 | 78.33 | 80.83 |
| **AR6** | 86.18 | 82.27 | 85.18 | 87.18 |

*Figure 4.2.1) Accuracy of best learners when aggregated in the above ensembles (Singh & Verma, 2018)*

While little to no work in the field of single classifiers (even Neural Networks) has been done on classifying multiple bugs from non-buggy data, the work in the Offside paper by Briem et al. as late as 2020 is comparable in that both this paper and their work involved artificially inducing bugs in non-buggy data to train classifiers to catch real world bugs. However, it must be noted that the work done by Briem et al. only covered a specific category of bugs unlike the work done in this paper which covered a much larger assortment. Nevertheless, the best performing Neural Network from the work done in this paper achieved comparable levels of accuracy at 66.7% on the test dataset in comparison to the 77.3% presented by Briem et al. on only a singular category of bugs.



| Statement type | Total | Accuracy | Recall | Precision | F1 |
|---|---|---|---|---|---|
| If | 49,418 | 0.72 | 0.6666 | 0.7462 | 0.7042 |
| For | 39,018 | 0.8723 | 0.8565 | 0.8844 | 0.8702 |
| While | 5,718 | 0.7272 | 0.6691 | 0.757 | 0.7104 |
| Return | 3,558 | 0.7476 | 0.6931 | 0.7779 | 0.7331 |
| Ternary | 3,114 | 0.6574 | 0.5466 | 0.7021 | 0.6147 |
| Method | 1,954 | 0.6592 | 0.5916 | 0.684 | 0.6345 |
| Assert | 608 | 0.6135 | 0.5164 | 0.6408 | 0.5719 |
| Do | 598 | 0.689 | 0.5886 | 0.7364 | 0.6543 |
| Var. decl. | 544 | 0.6581 | 0.5368 | 0.7087 | 0.6109 |
| Assign | 336 | 0.5982 | 0.4464 | 0.641 | 0.5263 |
| Expression | 72 | 0.6111 | 0.4167 | 0.6818 | 0.5172 |
| Obj. creation | 20 | 0.55 | 0.4 | 0.5714 | 0.4706 |
| Total | 104,958 | 0.7733 | 0.7303 | 0.799 | 0.7631 |

*Figure 4.2.2) Performance metrics of Offside approach on different statement types*

## 4.2) Analysis of Work Done

Given the results presented in Section 3 and the comparisons made in Section 4.1, this section will only specifically address the research questions presented in Section 1:

1. How well do ensemble methods from Scikit-learn do when attempting to classify non-buggy data vs. buggy data in Python code when a) multiple bug types need to be detected and classified and b) the bugs are artificially introduced?
   a. Even the best ensemble methods from Scikit-learn do not achieve beyond 23% overall classification accuracy on the variety of bugs induced into Python code in this paper with most of them averaging out at ~20%.
2. How well do neural networks from Tensorflow do when attempting to classify non-buggy data vs. buggy data in Python code when a) multiple bug types need to be detected and classified and b) the bugs are artificially introduced?
   a. Neural Networks from Tensorflow are able to achieve up to 75% classification accuracy on the validation dataset and 66.7% accuracy on the test dataset indicating that they do perform well when attempting to classify different bug types from non-buggy code in the current setup.
3. Given the problem being solved, which ML method performs the best and how do they compare against each other? Can Keras' Autotuner substantially boost the performance of the best methods?
   a. Neural Networks far outperform other methods in terms of classification accuracy and training time indicating that they are able to gain a deeper understanding of the code features and thereby what constitutes a bug. The combination of repeated dense layers with convolutional units is likely instrumental in helping it combine features appropriately instead of ensemble methods that seem unable to combine decision boundaries in a meaningful way at present. The best single classifier methods from scikit-learn were the LinearSVC and DecisionTreeClassifier and the best ensemble methods were the GradientBoostingClassifier and the ExtraTreesClassifier. Keras' AutoTuner was able to substantially improve the performance of Tensorflow's Neural Networks although the same was not true for the scikit-learn ensemble methods indicating that these methods need to be optimized in a different way for this problem.
4. Is the performance of the best classifier comparable to industry and research standards? Are there any takeaways from the work done that could be applied to future research in the field?
   a. The performance of the best Neural Network is indeed comparable to both industry and research standards especially given the greater scope of the problem being solved in this paper. This comparison has been extensively made in Section 4.1. The key takeaways from this paper in the present are the unique input pipelines introduced as well as the usage of Keras' AutoTuner to adapt a Neural Network solution to this problem.

## 5) Self-Evaluation

For clarity of thought, this section will be answered in the following categories:

1. What I learned:
   a. How to pose a Computer Science problem from current academic literature and formulate a solution for the same
   b. How to research tools and modules that might help simplify the solution and avoid repeating already-optimized industry-standard code
   c. How to pre-process, ingest and manipulate input data to efficiently utilize a wide variety of ML tools
   d. How to use ML tools from Scikit-learn, Tensorflow, and Keras in great depth to solve a specific problem
2. What I found challenging:
   a. How to alter the scope of the project according to the resources available

   b. How to balance between over-engineering a solution or attempting a different solution entirely
   c. How to select the right combination of tools to use - especially given the inability to test and try every combination
3. What I accomplished:
   a. Developed an input pipeline that artificially induces bugs in Python code to create novel datasets
   b. Learned how to adapt existing ML solutions to the rapidly growing problem of bug detection
   c. Developed a Neural Network powered bug detector that was able to achieve a high level of accuracy for bug classification in spite of having to classify multiple bug types
   d. Optimized the aforementioned Neural Network using Keras AutoTune to achieve validation and test accuracy that are comparable with research standard bug detectors

## 6) References

This section contains a list of any references used in building the work that was done in this paper or a comparison thereof.

### 6.1) Software Reused

1. https://scikit-learn.org/stable/supervised_learning.html
2. https://keras-team.github.io/keras-tuner/documentation/tuners/
3. https://www.tensorflow.org/api_docs/python/tf/keras/layers
4. https://datascienceplus.com/multi-class-text-classification-with-scikit-learn/

### 6.2) Comparable Work

1. Aleem, S., Capretz, L. F., & Ahmed, F. (2015). Benchmarking machine learning techniques for software defect detection. International Journal of Software Engineering & Applications, 6(3), 11-23. doi:10.5121/ijsea.2015.6302
2. Singh, P., & Verma, S. (2018). Multi-Classifier Model for Software Fault Prediction. The International Arab Journal of Information Technology, 15(5). Retrieved February 18, 2021, from https://iajit.org/PDF/September%202018,%20No.%205/10219.pdf
3. Briem, J. A., Smit, J., Sellik, H., Rapoport, P., Gousios, G., & Aniche, M. (2020). Offside. Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. doi:10.1145/3387940.3391464

### 6.3) Other References

1. Liu, S. (2021, January 29). Worldwide it enterprise software SPENDING 2009-2022. Retrieved February 24, 2021, from https://www.statista.com/statistics/203428/total-enterprise-software-revenue-forecast/
2. Liu, S. (2020, March 09). QA and testing budget allocation 2012-2019. Retrieved February 24, 2021, from https://www.statista.com/statistics/500641/worldwide-qa-budget-allocation-as-percent-it-spend/
3. Rafaela Azevedo. (2018, April 27). What is the cost of a bug? Retrieved April 18, 2021, from https://azevedorafaela.com/2018/04/27/what-is-the-cost-of-a-bug/#:~:text=If%20it's%20not%20found%20until,US%20economy%20%2459.5%20billion%20annually
4. Merrill, C. (2020, April 16). The incredible growth of python: Zibtek blog. Retrieved April 18, 2021, from https://www.zibtek.com/blog/the-incredible-growth-of-python/#:~:text=Since%20last%20year%2C%20Python%20has,easy%20to%20learn%20and%20understand.

# 7) Appendix

This section contains data that is not immediately relevant to the paper or is too long to be included in the main body.

## 7.1) Project Deliverables

All the data and code that form the basis of the work presented in this paper can be found at https://github.com/andybirla/ml_bug_detection. This is a public Github repository and therefore should be accessible by any user.

## 7.2) Additional Data

This section contains all the additional classification data from running the various models.

### 7.2.1) Single Classifier Classification Reports

This section contains the classification reports generated from running the Single Classifiers of Scikit on the validation dataset.

```
LinearSVC
Validation Accuracy: 0.19319754795333202
              precision    recall  f1-score   support
           0       0.09      0.12      0.11      1024
           1       0.08      0.08      0.08      1062
           2       0.26      0.16      0.20       408
           3       0.41      0.47      0.44       299
           4       0.00      0.00      0.00       276
           5       0.30      0.33      0.32       498
           6       0.51      0.63      0.57       468
           7       0.38      0.35      0.36       124
           8       0.08      0.07      0.07       748
           9       0.06      0.05      0.06       117
          10       0.00      0.00      0.00        33

    accuracy                           0.19      5057
   macro avg       0.20      0.21      0.20      5057
weighted avg       0.18      0.19      0.19      5057
```

```
MultinomialNB
Validation Accuracy: 0.11983389361281392
              precision    recall  f1-score   support
           0       0.11      0.24      0.15      1024
           1       0.10      0.16      0.12      1062
           2       0.12      0.02      0.03       408
           3       0.34      0.07      0.12       299
           4       0.06      0.01      0.01       276
           5       0.29      0.19      0.23       498
           6       0.23      0.08      0.12       468
           7       1.00      0.03      0.06       124
           8       0.05      0.03      0.04       748
           9       0.00      0.00      0.00       117
          10       0.00      0.00      0.00        33

    accuracy                           0.12      5057
```

```
   macro avg       0.21      0.08      0.08      5057
weighted avg       0.16      0.12      0.11      5057
```

```
LogisticRegression
Validation Accuracy: 0.15780106782677478
              precision    recall  f1-score   support
           0       0.09      0.16      0.12      1024
           1       0.08      0.09      0.08      1062
           2       0.26      0.10      0.15       408
           3       0.42      0.31      0.36       299
           4       0.00      0.00      0.00       276
           5       0.30      0.29      0.30       498
           6       0.48      0.34      0.40       468
           7       0.89      0.13      0.23       124
           8       0.09      0.08      0.08       748
           9       0.15      0.15      0.15       117
          10       0.00      0.00      0.00        33

    accuracy                           0.16      5057
   macro avg       0.25      0.15      0.17      5057
weighted avg       0.19      0.16      0.16      5057
```

```
DecisionTreeClassifier
Validation Accuracy: 0.2078307296816294
              precision    recall  f1-score   support
           0       0.18      0.59      0.27      1024
           1       0.17      0.08      0.11      1062
           2       0.49      0.08      0.14       408
           3       0.31      0.18      0.23       299
           4       0.06      0.00      0.01       276
           5       0.37      0.21      0.27       498
           6       0.39      0.18      0.25       468
           7       0.37      0.20      0.26       124
           8       0.19      0.08      0.11       748
           9       0.07      0.03      0.04       117
          10       0.00      0.00      0.00        33

    accuracy                           0.21      5057
   macro avg       0.24      0.15      0.15      5057
weighted avg       0.24      0.21      0.18      5057
```

```
KNeighborsClassifier
Validation Accuracy: 0.180344077516314
              precision    recall  f1-score   support
           0       0.14      0.21      0.16      1024
           1       0.14      0.13      0.13      1062
           2       0.16      0.11      0.13       408
           3       0.31      0.27      0.29       299
           4       0.15      0.08      0.10       276
           5       0.31      0.24      0.27       498
           6       0.21      0.45      0.29       468
           7       0.58      0.15      0.24       124
           8       0.16      0.09      0.11       748
           9       0.15      0.08      0.10       117
          10       0.00      0.00      0.00        33

    accuracy                           0.18      5057
```

```
   macro avg       0.21      0.16      0.17      5057
weighted avg       0.19      0.18      0.17      5057
```

## 7.2.2) Ensemble Classifier Classification Reports

```
RandomForestClassifier
0.19715246193395294
              precision    recall  f1-score   support
           0       0.11      0.14      0.12      1024
           1       0.10      0.08      0.09      1062
           2       0.28      0.22      0.25       408
           3       0.34      0.45      0.39       299
           4       0.04      0.03      0.03       276
           5       0.29      0.32      0.31       498
           6       0.43      0.66      0.52       468
           7       0.23      0.20      0.22       124
           8       0.07      0.05      0.06       748
           9       0.03      0.03      0.03       117
          10       0.00      0.00      0.00        33

    accuracy                           0.20      5057
   macro avg       0.18      0.20      0.18      5057
weighted avg       0.17      0.20      0.18      5057
```

```
AdaBoostClassifier
0.19517500494364248
              precision    recall  f1-score   support
           0       0.20      0.92      0.33      1024
           1       0.14      0.03      0.04      1062
           2       0.00      0.00      0.00       408
           3       0.00      0.00      0.00       299
           4       0.50      0.00      0.01       276
           5       0.60      0.01      0.01       498
           6       0.00      0.00      0.00       468
           7       0.00      0.00      0.00       124
           8       0.16      0.01      0.02       748
           9       0.05      0.02      0.03       117
          10       0.08      0.12      0.10        33

    accuracy                           0.20      5057
   macro avg       0.16      0.10      0.05      5057
weighted avg       0.18      0.20      0.08      5057
```

```
BaggingClassifier
0.16688741721854305
              precision    recall  f1-score   support
           0       0.14      0.21      0.17       159
           1       0.12      0.13      0.12       146
           2       0.17      0.06      0.09        81
           3       0.29      0.27      0.28        60
           4       0.00      0.00      0.00        42
           5       0.32      0.21      0.26        84
           6       0.21      0.36      0.26        67
           7       0.31      0.20      0.24        20
           8       0.09      0.09      0.09        78
           9       0.00      0.00      0.00        11
          10       0.00      0.00      0.00         7
```

```
    accuracy                           0.17       755
   macro avg       0.15      0.14      0.14       755
weighted avg       0.16      0.17      0.16       755

ExtraTreesClassifier
0.17880794701986755
              precision    recall  f1-score   support
           0       0.05      0.06      0.05       159
           1       0.04      0.05      0.05       146
           2       0.26      0.16      0.20        81
           3       0.51      0.50      0.50        60
           4       0.00      0.00      0.00        42
           5       0.24      0.26      0.25        84
           6       0.36      0.64      0.46        67
           7       0.31      0.50      0.38        20
           8       0.00      0.00      0.00        78
           9       0.00      0.00      0.00        11
          10       0.00      0.00      0.00         7

    accuracy                           0.18       755
   macro avg       0.16      0.20      0.17       755
weighted avg       0.16      0.18      0.16       755


GradientBoostingClassifier
0.20662251655629138
              precision    recall  f1-score   support
           0       0.24      0.18      0.21       159
           1       0.15      0.31      0.20       146
           2       0.22      0.22      0.22        81
           3       0.34      0.25      0.29        60
           4       0.10      0.05      0.06        42
           5       0.32      0.21      0.26        84
           6       0.35      0.36      0.36        67
           7       0.14      0.20      0.16        20
           8       0.03      0.01      0.02        78
           9       0.00      0.00      0.00        11
          10       0.00      0.00      0.00         7

    accuracy                           0.21       755
   macro avg       0.17      0.16      0.16       755
weighted avg       0.21      0.21      0.20       755


VotingClassifier
0.1403973509933775
              precision    recall  f1-score   support
           0       0.08      0.11      0.09       159
           1       0.08      0.18      0.11       146
           2       0.27      0.04      0.07        81
           3       0.55      0.28      0.37        60
           4       0.00      0.00      0.00        42
           5       0.26      0.23      0.24        84
           6       0.38      0.31      0.34        67
           7       1.00      0.10      0.18        20
           8       0.00      0.00      0.00        78
           9       0.00      0.00      0.00        11
          10       0.00      0.00      0.00         7

    accuracy                           0.14       755
   macro avg       0.24      0.11      0.13       755
weighted avg       0.19      0.14      0.14       755
```

### 7.2.3) Neural Network Trials

The text below shows the 10 best trials when running the final neural network.

```
Results summary
Results in ./untitled_project
Showing 10 best trials
Objective(name='val_accuracy', direction='max')
Trial summary
Hyperparameters:
units: 224
units_1: 24
activation_func: 2
additional_dense_layer: 2
units_2: 40
activation_func_2: 1
activation_func_3: 2
strides: 1
kernel_size: 5
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.5
tuner/epochs: 30
tuner/initial_epoch: 10
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 3bb19afd3f0ab9ebcbedd28483213ece
Score: 0.7326139211654663
Trial summary
Hyperparameters:
units: 208
units_1: 24
activation_func: 1
additional_dense_layer: 1
units_2: 40
activation_func_2: 1
activation_func_3: 1
strides: 4
kernel_size: 5
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.5
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 3
tuner/round: 2
tuner/trial_id: 642fae39802bdf4b933166c6ba03cf92
Score: 0.7314148545265198
Trial summary
Hyperparameters:
units: 256
units_1: 28
activation_func: 2
additional_dense_layer: 1
units_2: 32
activation_func_2: 2
```

```
activation_func_3: 1
strides: 1
kernel_size: 3
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.1
tuner/epochs: 30
tuner/initial_epoch: 10
tuner/bracket: 3
tuner/round: 3
tuner/trial_id: 3621771eed7a749e8f784a63c189e41f
Score: 0.7194244861602783
Trial summary
Hyperparameters:
units: 256
units_1: 28
activation_func: 2
additional_dense_layer: 1
units_2: 32
activation_func_2: 2
activation_func_3: 1
strides: 1
kernel_size: 3
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.1
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 3
tuner/round: 2
tuner/trial_id: dcfa922911a3350e538897aa7cfdea33
Score: 0.7182254195213318
Trial summary
Hyperparameters:
units: 208
units_1: 32
activation_func: 2
additional_dense_layer: 2
units_2: 36
activation_func_2: 2
activation_func_3: 2
strides: 1
kernel_size: 3
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.4
tuner/epochs: 30
tuner/initial_epoch: 10
tuner/bracket: 2
tuner/round: 2
tuner/trial_id: 4b184880e2b66ee42ad8d7abb855c4d3
Score: 0.7170263528823853
Trial summary
Hyperparameters:
units: 128
units_1: 32
activation_func: 1
additional_dense_layer: 1
units_2: 56
activation_func_2: 1
```

```
activation_func_3: 1
strides: 3
kernel_size: 5
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.2
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 3
tuner/round: 2
tuner/trial_id: de397c50d475a31f03726491a9640583
Score: 0.714628279209137
Trial summary
Hyperparameters:
units: 224
units_1: 24
activation_func: 2
additional_dense_layer: 2
units_2: 40
activation_func_2: 1
activation_func_3: 2
strides: 1
kernel_size: 5
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.5
tuner/epochs: 10
tuner/initial_epoch: 0
tuner/bracket: 1
tuner/round: 0
Score: 0.714628279209137
Trial summary
Hyperparameters:
units: 176
units_1: 20
activation_func: 2
additional_dense_layer: 2
units_2: 40
activation_func_2: 1
activation_func_3: 1
strides: 2
kernel_size: 5
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.1
tuner/epochs: 30
tuner/initial_epoch: 10
tuner/bracket: 3
tuner/round: 3
tuner/trial_id: 258c739c785cfbb266e8a67f4e86ea1d
Score: 0.7122302055358887
Trial summary
Hyperparameters:
units: 160
units_1: 28
activation_func: 1
additional_dense_layer: 2
units_2: 40
activation_func_2: 2
activation_func_3: 1
```

```
strides: 2
kernel_size: 7
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.1
tuner/epochs: 30
tuner/initial_epoch: 10
tuner/bracket: 3
tuner/round: 3
tuner/trial_id: 6a4d2d53f47c0e00c41e11c49ecd90df
Score: 0.7110311985015869
Trial summary
Hyperparameters:
units: 160
units_1: 28
activation_func: 1
additional_dense_layer: 2
units_2: 40
activation_func_2: 2
activation_func_3: 1
strides: 2
kernel_size: 7
learning_rate: 0.001
pooling_layer: 1
dropout_rate: 0.1
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 3
tuner/round: 2
tuner/trial_id: 28d3e66dfc37da23671a63661b09c381
Score: 0.7074340581893921
```

## 7.2.4) Additional Comparable Work

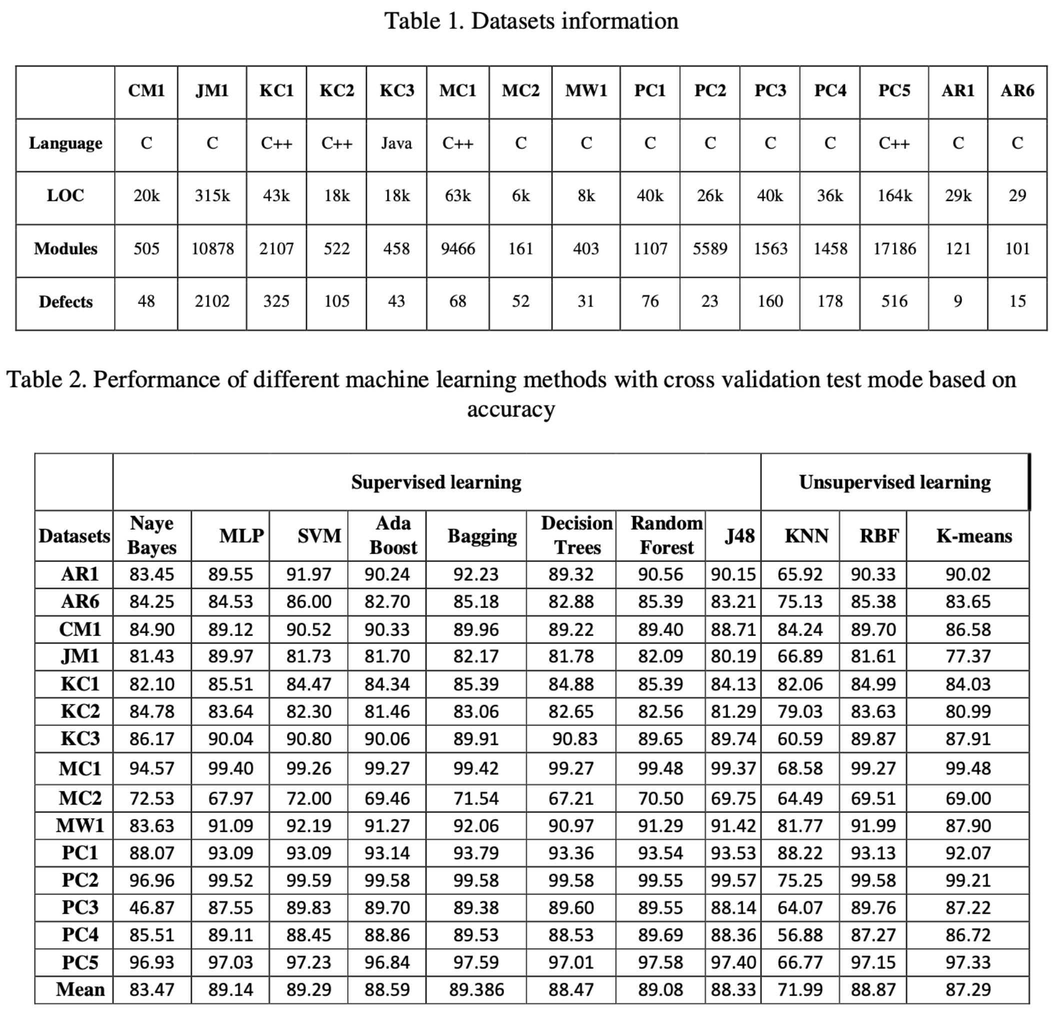


Table 1. Datasets information

| | CM1 | JM1 | KC1 | KC2 | KC3 | MC1 | MC2 | MW1 | PC1 | PC2 | PC3 | PC4 | PC5 | AR1 | AR6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Language** | C | C | C++ | C++ | Java | C++ | C | C | C | C | C | C | C++ | C | C |
| **LOC** | 20k | 315k | 43k | 18k | 18k | 63k | 6k | 8k | 40k | 26k | 40k | 36k | 164k | 29k | 29 |
| **Modules** | 505 | 10878 | 2107 | 522 | 458 | 9466 | 161 | 403 | 1107 | 5589 | 1563 | 1458 | 17186 | 121 | 101 |
| **Defects** | 48 | 2102 | 325 | 105 | 43 | 68 | 52 | 31 | 76 | 23 | 160 | 178 | 516 | 9 | 15 |

Table 2. Performance of different machine learning methods with cross validation test mode based on accuracy

| | **Supervised learning** | | | | | | | | **Unsupervised learning** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Datasets** | **Naye Bayes** | **MLP** | **SVM** | **Ada Boost** | **Bagging** | **Decision Trees** | **Random Forest** | **J48** | **KNN** | **RBF** | **K-means** |
| **AR1** | 83.45 | 89.55 | 91.97 | 90.24 | 92.23 | 89.32 | 90.56 | 90.15 | 65.92 | 90.33 | 90.02 |
| **AR6** | 84.25 | 84.53 | 86.00 | 82.70 | 85.18 | 82.88 | 85.39 | 83.21 | 75.13 | 85.38 | 83.65 |
| **CM1** | 84.90 | 89.12 | 90.52 | 90.33 | 89.96 | 89.22 | 89.40 | 88.71 | 84.24 | 89.70 | 86.58 |
| **JM1** | 81.43 | 89.97 | 81.73 | 81.70 | 82.17 | 81.78 | 82.09 | 80.19 | 66.89 | 81.61 | 77.37 |
| **KC1** | 82.10 | 85.51 | 84.47 | 84.34 | 85.39 | 84.88 | 85.39 | 84.13 | 82.06 | 84.99 | 84.03 |
| **KC2** | 84.78 | 83.64 | 82.30 | 81.46 | 83.06 | 82.65 | 82.56 | 81.29 | 79.03 | 83.63 | 80.99 |
| **KC3** | 86.17 | 90.04 | 90.80 | 90.06 | 89.91 | 90.83 | 89.65 | 89.74 | 60.59 | 89.87 | 87.91 |
| **MC1** | 94.57 | 99.40 | 99.26 | 99.27 | 99.42 | 99.27 | 99.48 | 99.37 | 68.58 | 99.27 | 99.48 |
| **MC2** | 72.53 | 67.97 | 72.00 | 69.46 | 71.54 | 67.21 | 70.50 | 69.75 | 64.49 | 69.51 | 69.00 |
| **MW1** | 83.63 | 91.09 | 92.19 | 91.27 | 92.06 | 90.97 | 91.29 | 91.42 | 81.77 | 91.99 | 87.90 |
| **PC1** | 88.07 | 93.09 | 93.09 | 93.14 | 93.79 | 93.36 | 93.54 | 93.53 | 88.22 | 93.13 | 92.07 |
| **PC2** | 96.96 | 99.52 | 99.59 | 99.58 | 99.58 | 99.58 | 99.55 | 99.57 | 75.25 | 99.58 | 99.21 |
| **PC3** | 46.87 | 87.55 | 89.83 | 89.70 | 89.38 | 89.60 | 89.55 | 88.14 | 64.07 | 89.76 | 87.22 |
| **PC4** | 85.51 | 89.11 | 88.45 | 88.86 | 89.53 | 88.53 | 89.69 | 88.36 | 56.88 | 87.27 | 86.72 |
| **PC5** | 96.93 | 97.03 | 97.23 | 96.84 | 97.59 | 97.01 | 97.58 | 97.40 | 66.77 | 97.15 | 97.33 |
| **Mean** | 83.47 | 89.14 | 89.29 | 88.59 | 89.386 | 88.47 | 89.08 | 88.33 | 71.99 | 88.87 | 87.29 |

*Figure 7.2.4.1) Performance of ML aggregator methods on different datasets (*Aleem, S., Capretz, L. F., & Ahmed, F. (2015)