symmetry

MDPI

Article

Automatic Classification of Equivalent Mutants in Mutation Testing of Android Applications

Muhammad Bello Kusharki [1], Sanjay Misra [2,*], Bilkisu Muhammad-Bello [3], Ibrahim Anka Salihu [3] and Bharti Suri [4]

[1] Department of Information and Communication Technology, National Defence College, Abuja 900128, Nigeria; mbkusharki@ndc.gov.ng
[2] Department of Computer Science and Communication, Østfold University College, P.O. Box 700, NO-1757 Halden, Norway
[3] Software Engineering & Information Technology Department, Nile University of Nigeria, Abuja 900001, Nigeria; bilkisu.muhammad-bello@nileuniversity.edu.ng (B.M.-B.); ibrahim.salihu@nileuniversity.edu.ng (I.A.S.)
[4] University School of Information and Communication Technology, GGS Indraprastha University, Delhi 110078, India; bhartisuri@ipu.ac.in
* Correspondence: sanjay.misra@hiof.no

Abstract: Software and symmetric testing methodologies are primarily used in detecting software defects, but these testing methodologies need to be optimized to mitigate the wasting of resources. As mobile applications are becoming more prevalent in recent times, the need to have mobile applications that satisfy software quality through testing cannot be overemphasized. Testing suites and software quality assurance techniques have also become prevalent, which underscores the need to evaluate the efficacy of these tools in the testing of the applications. *Mutation testing* is one such technique, which is the process of injecting small changes into the software under test (SUT), thereby creating *mutants*. These mutants are then tested using mutation testing techniques alongside the SUT to determine the effectiveness of test suites through mutation scoring. Although mutation testing is effective, the cost of implementing it, due to the problem of equivalent mutants, is very high. Many research works gave varying solutions to this problem, but none used a standardized dataset. In this research work, we employed a standard mutant dataset tool called MutantBench to generate our data. Subsequently, an Abstract Syntax Tree (AST) was used in conjunction with a tree-based convolutional neural network (TBCNN) as our deep learning model to automate the classification of the equivalent mutants to reduce the cost of mutation testing in software testing of android applications. The result shows that the proposed model produces a good accuracy rate of 94%, as well as other performance metrics such as recall (96%), precision (89%), F1-score (92%), and Matthew's correlation coefficients (88%) with fewer False Negatives and False Positives during testing, which is significant as it implies that there is a decrease in the risk of misclassification.

Keywords: software testing; artificial intelligence; mutation testing; android applications; tree-based convolutional neural networks

Citation: Kusharki, M.B.; Misra, S.; Muhammad-Bello, B.; Salihu, I.A.; Suri, B. Automatic Classification of Equivalent Mutants in Mutation Testing of Android Applications. *Symmetry* **2022**, *14*, 820. https://doi.org/10.3390/sym14040820

Academic Editor: SeongKi Kim

Received: 23 March 2022
Accepted: 10 April 2022
Published: 14 April 2022

1. Introduction

The identification of equivalent mutants is a crucial task in mutation testing for optimizing the cost of implementing the mutation testing. Strategies such as symmetric testing have been promising, but it requires the incorporation of other techniques and methodologies for the effective classification of equivalent mutants. The growing number of mobile devices, such as iPods, smartphones, androids, and tablets, resulted in the massive need for mobile solutions in today's world, which has increased the demand for mobile software applications. This huge pressure is forcing many applications to reach the market with significant faults, which often results in failures [1], thereby necessitating the need for

adequate software quality testing. Software testing is an aspect of agile programming [2] aimed at checking if an application meets the software requirement and if it will satisfy the needs of all. The tools that are employed in testing the software under test (SUT) are equally developing rapidly due to the fast-changing world. This underscores the need for techniques that will test these test suites.

To achieve this all-important software quality for all test suites, an effective software testing framework is required, and *Mutation Testing* is one of the most effective techniques [3]. Mutation Testing is a very effective technique for the testing of test suites and for generating, prioritizing, and selecting test cases through the utilization of mutation operators [4]. The SUT is mutated through the use of mutation operators to change some part of the source codes, thereby increasing the chances of the test suites giving two different outputs between the SUT and the Mutant program. If the mutants are killed, then we compute the mutation score [5], which is the basis for evaluating the test suites. The issue with this technique is the process involved in identifying the equivalent mutants within the mutant program after the mutants are killed because the mutation score computation relies heavily on the identification of these types of mutants. Equivalent mutants are key to computing the mutation score, and when the equivalent mutant code is not detected, the mutation score (*MS*) will be low, and that will imply that the mutation testing is not effective. The mutation score formula is used for the computation of the effectiveness of the mutation testing technique. Equation (1) below highlights the *MS* formula, where *P* is the program under test, *T* is the test suite, *K* is the number of killed mutants, *M* is the number of generated mutants, and lastly, *E* is the number of equivalent mutants. Research has been carried out towards solving this equivalent mutant problem (EMP), including machine learning-related approaches.

$$MS(P,T) = \frac{K}{(M-E)} \tag{1}$$

The application of Machine Learning (ML) has helped in all strata of the computing world, thereby creating huge research possibilities in the area of software quality which led to the proposition of different automation techniques for solving problems. Equally, deep learning has further redefined the automation of problems in software testing [6], where Abstract Syntax Tree (AST) techniques are combined with great models and techniques to handle natural language processing. These have helped in the proposition of models that automate the EMP [7] with promising results, which subsequently reduced the cost of mutation testing. Although novel models have been proposed and implemented, there is little research in the area of equivalent mutant testing for android applications. Since mutation testing is domain-specific [8], the desktop solution will not necessarily work for the mobile application domain.

From the above, we can deduce that equivalent mutant classification is key for effective mutation testing. Equivalent mutant classification will also lead to a reduction in the cost involved in this software testing technique because obtaining a very good mutation score is key to this framework (see Equation (1) below) [9]. Hence, this research aims to automate the classification of EMP in mutation testing of android applications using a deep learning model based on the Tree-based Convolutional Neural Network (TBCNN) architecture for an effective reduction in the cost of implementing the mutation testing. To achieve this aim, we generated a standard mutant dataset using the standardized tool proposed by [10]. Subsequently, as our first contribution, we developed a model based on TBCNN architecture to automate the classification of the equivalent mutant; secondly, we implemented and validated our model using cross-validation.

The current state-of-the-art methods in solving EMP for automatic classification of equivalent mutants were reviewed and evaluated against our research objectives which raised the following research questions: (1) Will the proposed deep learning model enhance the use of mutation testing techniques in the testing of android applications? (2) What are the benefits of the classification of equivalent mutants in the mutation testing of software

under test (SUT)? (3) Will automated identification of equivalent mutants solve all the problems associated with the high cost of mutation testing in android applications?

The remaining sections of this paper are structured as follows: Section 2 details the related works; Section 3 details our materials and methods; the implementation, results, and discussions are detailed in Section 4; Section 5 concludes the paper.

2. Related Works

Mutation testing, a powerful testing tool, is a technique of assessing and improving the quality of test suites [11] and is carried out through the creation of a modified version of a program, called a mutant, and then applying mutation operators that make small changes to the software by either simulating faults or guiding the tester to edge cases [12]. During the testing, testers are expected to find or design tests that cause these mutants to fail, that is, behave differently from the SUT. If a test case causes a mutant to fail, then that mutant is said to be *killed*; otherwise, the mutant remains *alive*. Mutation can be used to help testers design high-quality tests or to evaluate and improve existing test suites. Mutation testing technique has empirically shown that it is a very strong testing tool compared to control flow-based testing and data flow-based testing [13], but it generally has less usage due to the high cost of implementation as a result of problems, such as the equivalent mutant problem, that affect mutation score percentile.

Recently, a lot has been achieved in the area of research that effectively aids the mutation testing of android apps [1,5,12]. To adequately certify the test suites that these studies recommended, the mutation technique is used to determine the fault detection capabilities of these tools. In mutation testing, it is expected for testers to find a test suite that will cause the test on the mutant to fail or behave differently from the SUT [11], and when this happens, the mutant is said to be *killed*, otherwise, it is *alive*. Empirical studies have shown that for mutation testing to be very effective, mutation operators need to be applied to a specific domain. In [14], the researchers analyzed some android faults within a group of 2062 such faults, and they further identified 38 mutation operators that they use to classify their stillborn/trivial mutants. This research achieved a 77% accuracy in bug taxonomy, but [15] recommended a tool, MDroid+, to use the same 38 operators in the generation of mutants and testing of test suites with 8000 mutants generated from over 50 applications. Although ref. [16], in their paper, highlighted the faults inherent in MDroid+, they further recommended the Monkey tool as a better tool for generating data. Performance mutation testing framework, as proposed by [3], introduced the traditional mutation models and emphasized the performance enhancement of tests which the paper by [4] used to evaluate their developed tool, AMOGA, with an emphasis on about five mutation operators which include *ICR* (*Inline Constant Replacement*), *NOI* (*Negative Operator Inversion*), *LCR* (*Logical Connector Replacement*), *AOR* (*Arithmetic Operator Replacement*), and *ROR* (*Relational Operator Replacement*). All the listed works of literature here show the importance of mutation testing to the android applications, but none handle the issue of the cost of implementing the technique as a result of equivalent mutants.

One of the first attempts to handle the equivalent mutant problem was [17], where they used the manual technique of hand-labeling the equivalent mutants; this work opened doors to the research into the automation of this classification task. In [18], the researchers proposed the use of an Abstract Syntax Tree (AST) with combined models of Tree-Based Convolutional Neural Network (TBCNN) and Support Vector Machine (SVM), and this gave an accuracy of 92%. Another paper used deep CNN (DCNN) for the same classification but returned an accuracy of 60% with a great focus on incremental learning [19]. The work of [8] reduced the mutant testing effort using their MuAPK 2.0 tool through the removal of dead code mutants with an accuracy of 90%, while [5] used mathematical constraints in constraint-based testing (CBT) theory and Random Forest to achieve 80% accuracy in their approach for the classification of equivalent mutants. Furthermore, refs. [2,20] used *K*-nearest neighbors (k-NN) and MutantDistiller, respectively, in their attempt to improve the classification of mutants so that it will enhance the use of mutation testing. In an attempt to blend neural networks and AST, ref [21] proposed the use of ASTNN, a novel

machine learning model that automatically classified equivalent mutants using two popular mutation operators and achieved an accuracy of 90%. All these papers did a good job towards the classification of equivalent mutants but have two things in common: they did not use a standard dataset based on the FAIR principle as proposed by [10], and they and very few other studies were in the domain of mobile applications. In addition, the use of the deep learning approach is also very minimal.

The use of TBCNN in natural language processing has become important because of the availability of ASTs and the capacity to handle both the semantic and syntactic nature of program source codes [22]. From combined AST, as used by [6], where they used the concept of the node and obtained an accuracy of 93% in the bug localization, to [21], which used TBCNN for incremental learning due to the hierarchical nature of the program source codes and with an accuracy of 87%, it can be deduced that TBCNN is a very good model for incremental learning and language processing. Furthermore, ref. [23] has opened more windows for some techniques that can be combined with ASTs and TBCNN to achieve very good accuracy in the classification of equivalent mutants. In this paper, drawing strength from all the above literature, we combined ASTs with TBCNN and incorporated Bi-GRU and LSTM to achieve our aim.

From these pieces of literature reviewed, we observed that many studies were conducted regarding equivalent mutation problems (EMP) but with limited research work within the mobile apps' domain. Furthermore, some literature that covered the automation of the classification of equivalent mutants used some novel techniques, such as the ASTNN [21] and MutantDistiller [20], but none handled the classification of the equivalent mutant from the mobile apps domain perspective or used tree-based convolutional neural networks. It could equally be deduced that TBCNN has given a high level of accuracy compared to other models, most especially when handling source codes of software programs as we intend to do [21]. In Section 2, we explained, in detail, the materials and methods used in solving the problem of automating the classification of equivalent mutants in the mutation testing of android applications.

3. Materials and Methods

Our research methodology was inspired by the research of [22] but with some modifications to accommodate our contribution in using a standardized dataset as recommended by the work of [10], based on the FAIR principle. Figure 1 depicts our research design methodology. It shows the entire process from data generation and data pre-processing with word embeddings training and sequence of blocks generation before feeding our TBCNN network (a model based on a tree-based convolutional neural network, as stipulated by [6,24,25] in their research paper, which gave them accuracy in the region of 85–90%, respectively), with the vector representation of the mutant dataset.

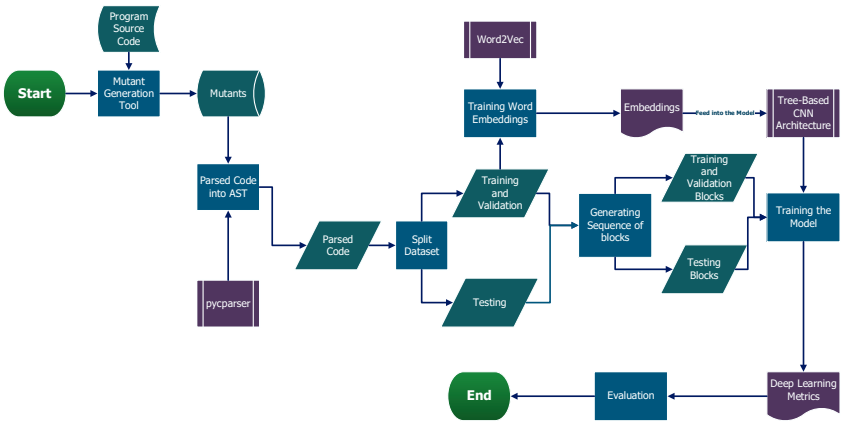

Figure 1. The methodology flowchart is deployed in this thesis.

3.1. Data Generation and Processing

The research in the area of mutation testing, based on the domain knowledge of the android apps, required the generation of mutants as the dataset for any further analysis, and we achieved that using the FAIR data principles. We sampled 30 programs' source codes in generating our dataset. We equally needed to carry out data processing so that we obtained a dataset in a format suitable for our model. As we elaborated, we used MutantBench to generate our dataset following the concept of [10] as postulated in Equation (2) below:

$$\sum_{p \in P} |p| \ * \ |M_p| \tag{2}$$

where P is the set of all the programs within the dataset and $|M_p|$ equals the mutants that are with the program. Even though mutation testing is deep in strategy, it comes with a high cost for data generation, most especially in source codes of android applications, as they are usually large. Though the solution is to store the filename of each of the equivalent mutants and present a method of the mutant generation where they are generated within the same set of mutants and with the same file, it is still tedious as it will require the running of the same generation tool with same attributes and possibly with same environments.

To solve this anomaly, we adopted the method of [26] but kept in mind the need to consider semantic and synthetic structures of the software under test (SUT) as proposed by [21]. In Figure 2 below, we present the process flow of our data generation using the *μjava* tool within MutantBench. We then pre-labeled our dataset as a *positive* class for the equivalent mutants and a *negative* class for the generated killed mutants, all from the same SUT. We limited our selection of mutant operators to absolute value insertions (ABS) [1,17] because it is widely used and covers a lot of codes. Several papers used other tools to generate combined mutants, which include equivalent mutant datasets [16–18,24,27].

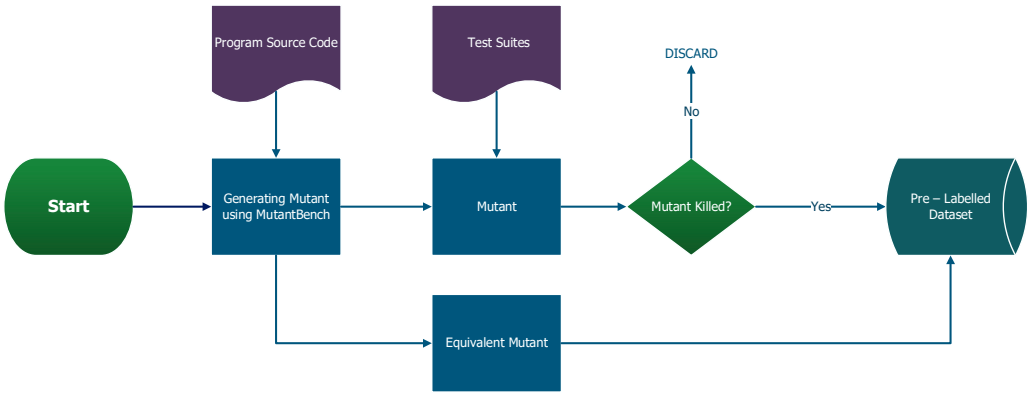

Figure 2. The Data Generation Dataflow Diagram.

We carried out the processing of our pre-labeled dataset by parsing it using pycparser, as our dataset was generated from C and C++ programs. The MutantBench being a flexible tool made this possible. The parsed dataset was then shuffled and split into training, validation, and test. The training and validation dataset was then trained with the popular *Word2Vec* dictionary, ref. [28], in an unsupervised way. Subsequently, the word embeddings were created using a skip-gram training algorithm with an embedding size of 128. Using these embeddings, we generated a sequence of blocks for the training, validation, and testing before feeding our model with these sub-datasets. The flowchart in Figure 3 below depicts the entire data processing flow.

3.2. Model Design and Training

The natural flow of programming languages is a tree-like structure, just like an AST representation, as shown in Figure 4a, which represents Equation (3). Our mutant dataset equally follows this structure; therefore, we adopted the use of the TBCNN model together with AST.

$$Int\ a = b + 3 \tag{3}$$

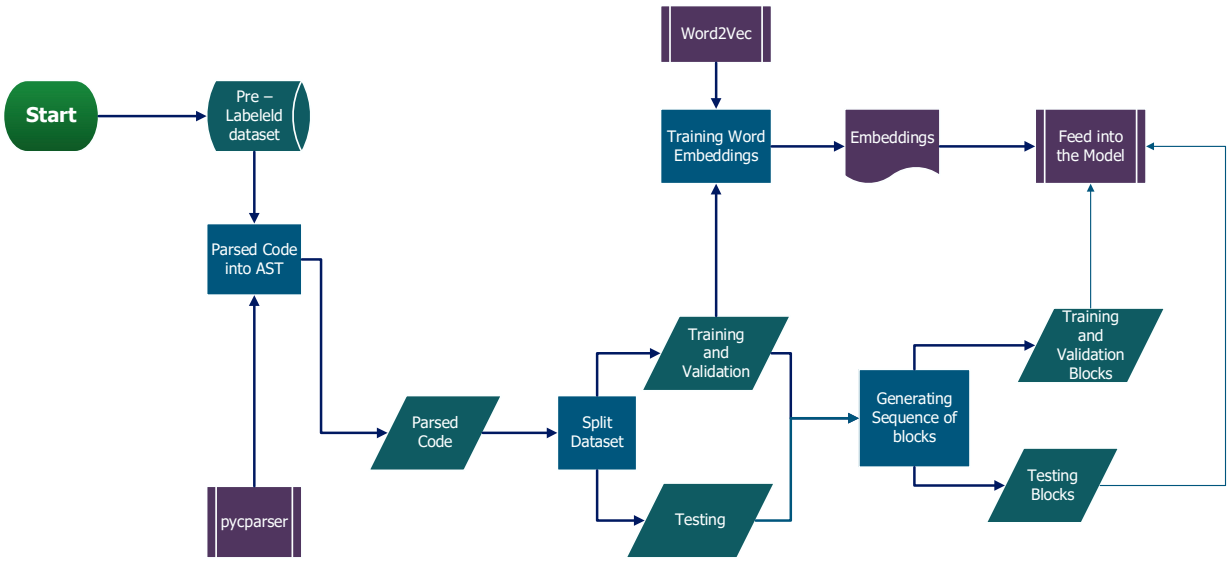

Figure 3. The Data Processing Dataflow Diagram.

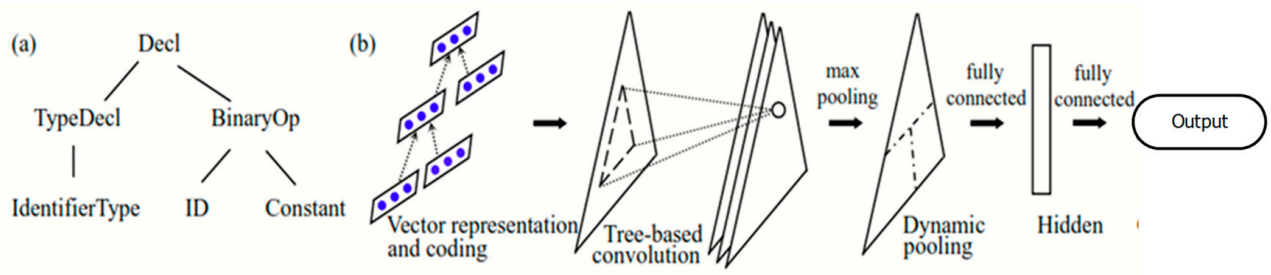

Figure 4. The Tree-Based Convolutional Neural Networks.

The model used for this classification is based on TBCNN, and we chose this architecture based on the inspiration of its hierarchical classifiers, where it comprises, multiple nodes connected in a tree-like pattern [22]. The TBCNN model has some initial layers of Convolutional Neural Networks (CNN) that learn some of the generic parts of the features hierarchically, classifying the classes through the use of the upper nodes. Subsequently, new hierarchies are developed to accommodate new classes [20,29].

In most models that handled the automatic classification of the equivalent mutants, features extraction is handled separately, but in TBCNN, it is conducted by the model itself [23]. The TBCNN begins as a single root node and then generates new hierarchies to accommodate new classes [22]. A typical TBCNN works as described below [7]:

1. Initially, the network is trained to classify data into N categories. The data from a new class is presented to the network, and the network then expands to accommodate the new class;
2. The network expands by adding a new leaf/branch node to the existing structure;
3. The goal of reducing training effort has two components: the number of weights updated and the number of examples, old or new, required for training;
4. Lastly, changes have been restricted to a new branch of the tree.

The TBNN is used for the extraction of features and the classification of the mutants into equivalent mutants or and nonequivalent mutants. Figure 4b shows the process flow, where for every non-leaf p and its children, $x_1, x_2, \ldots.., x_n$, we would then have Equation (4).

Each node in AST is placed as a real value vector, such that the features of the symbols are captured as vector representation through a coding criterion as proposed by [18].

$$vec(p) \approx \tanh\left(\sum_n l_n W_{code,n} \cdot vec(x_n) + b_{code}\right) \quad (4)$$

where $W_{code,n} \in \mathbb{R}^{N_f \times N_f}$ is the weight matrix that corresponds to the symbol x_n and $b_{code} \in \mathbb{R}^{N_f}$ is the bias, l_n which equals $\frac{\#leaves\ under\ x_n}{\#Leaves\ under\ p}$ is the coefficient of the weight. Because different nodes may have different numbers of branch nodes, that means $W_{code,n}$ is not fixed. Therefore, to solve this problem, we introduced the concept of the continuous binary tree where only two weight matrices, W^l_{code} and W^r_{code} , serve as model parameters. The weight, W_n, is the linear combination of the two-parameter matrices. The major closeness between $vec(p)$ and the coded vector is measured by the Euclidean distance equation:

$$d = \left\| vec(p) - \tanh\left(\sum_n l_n W_{code,n} \cdot vec(x_n) + b_{code}\right) \right\|_2^2 \quad (5)$$

After the pre-training of the feature vectors for every symbol and using the popular word embeddings, *Word2Vec* [27], we fed that, together with the trained embeddings, into the tree-based convolutional neural network architecture. Figure 4 shows the representation of the TBCNN and it clearly shows the nodes on the left as a representation of feature vectors of the symbols in AST. To further explain our techniques, the vectors were pre-trained based on embeddings, using BI-GRU [28] and max-pooling to pool the hidden states of the BI-GRU into a single vector representation source code. After the pre-training of the features of the vectors for every symbol, we fed our model with the pre-trained dataset together with the word embeddings. The entire model architecture and the vector presentation architectures are shown in Figures 5 and 6 below.

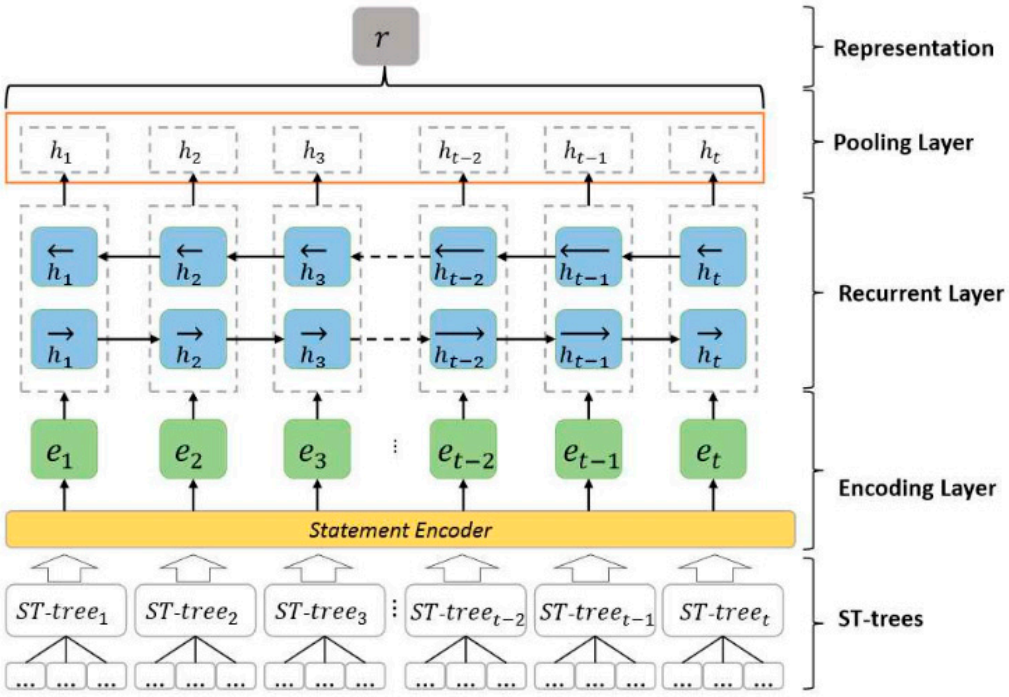

Figure 5. The Model Architecture Workflow.

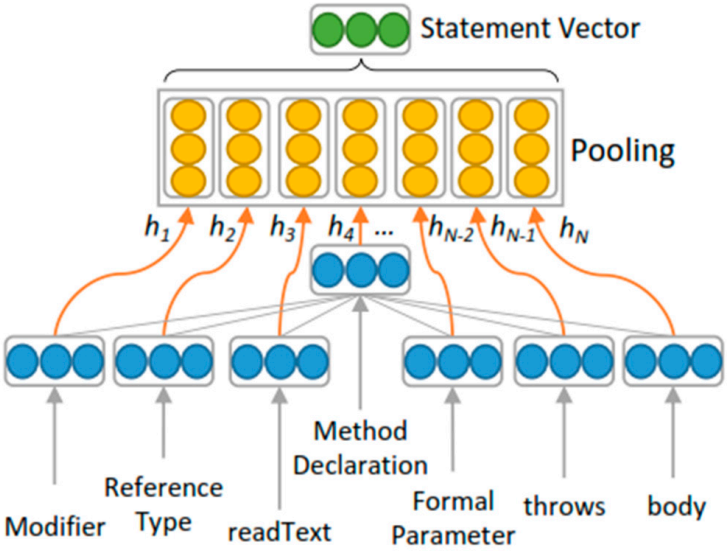

Figure 6. The Vector Representation.

Training and Validation

Since deep learning requires a large amount of data and our pre-labeled dataset constitutes only 52,000 samples, we handled the training and validation of the model through supervised learning and deployed the cross-validation training method in training our model. Cross-validation is a form of K—sampling method where the training is conducted with k = 10 and then we perform a series of training and validation steps with hyperparameter tuning using cross-entropy and AdaMax [29] as optimizers. To obtain very good accuracy, we used 10 epochs, a hidden dimension of 300, and a batch size of 32; this is good because it is updated frequently enough to promote effective training without drastically slowing down the process.

As part of training and parameter setting, we tried running our model on 3 different platforms: (1) Jupyter Notebook; (2) Google Colab; and (3) Amazon SageMaker Studio Lab. We found out that SageMaker gave the best "*Time Cost*", which is one of the two key problems we intended to solve. With AWS SageMaker, we ran our model with 153 s of time compared to Jupyter Notebook's 21,000 s and Google Colab's 1403 s. This clearly shows that the environment and tools are key players in improving the time cost of finding equivalent mutants in the mutation testing of android applications.

To evaluate our model, we used performance evaluation metrics, such as confusion matrix, recall, precision, F1-score, and Matthew's correlation coefficient (MCC). We set the confusion matrix as below:

1. True Positive (*TP*)—Mutant is equivalent, as anticipated by our model;
2. False Positive (*FP*)—Mutant is not equivalent, but our model predicted it was equivalent;
3. True Negative (*TN*)—Mutant is not equivalent, and our model predicted it was not equivalent;
4. False Negative (*FN*)—Mutant is equivalent, but our model predicted it was not equivalent.

The following equations were then used for the above-stated evaluation parameters. The table listed the software resources necessary for this model implementation and to achieve the minimum results like ours.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{6}$$

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

$$Precision = \frac{TP}{TP + FP} \tag{8}$$

$$\text{F1} - \text{Score} = 2 * \left(\frac{Precision * Recall}{Precision + Recall} \right) \tag{9}$$

Given that we used a binary classification in solving the problem of equivalent mutants, we further evaluated our model using Matthew's correlation coefficients (MCC). MCC is not affected by the issue of unbalanced datasets, ref. [30], as this matrix is a method of calculating the *Pearson product-moment correlation coefficient* between actual and predicted values (within the range of −1 as the worst value and +1 as the best value). The MCC is defined as:

$$\text{MCC} = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}} \tag{10}$$

MCC is the only binary classification rate that gives a more reliable statistical rate that yields a high score if the prediction performed well in all four confusion matrix categories (true positives, false negatives, true negatives, and false positives), according to the number of positive and negative items in the dataset [31].

Table 1 below shows the required software resources necessary for this model implementation.

Table 1. List of software resources.

Serial Number	Resources	Version
1	Android Studio	2020.3.1
2	MutantBench	2021.1.1
3	Gumtre	2.1.2
4	Python	3.10.1
5	Pandas	
6	Gensim	3.5.0
7	Pytorch	1.6.0
8	Dataloader	
9	Pycparser	2.18
10	Google Colab	-

4. Discussion of Results

In this section, we explain our model results and then compare them with other related research work as captured in our literature review. The absolute value insertion (ABS) mutation operator was chosen to run our model because it is the most widely used operator, and it will serve as the baseline for the future implementation of our model. Using this mutant operator and the MutantBench tool, we generated a standard dataset based on the FAIR principle [10], for both positive and negative classes as shown in Figure 7.

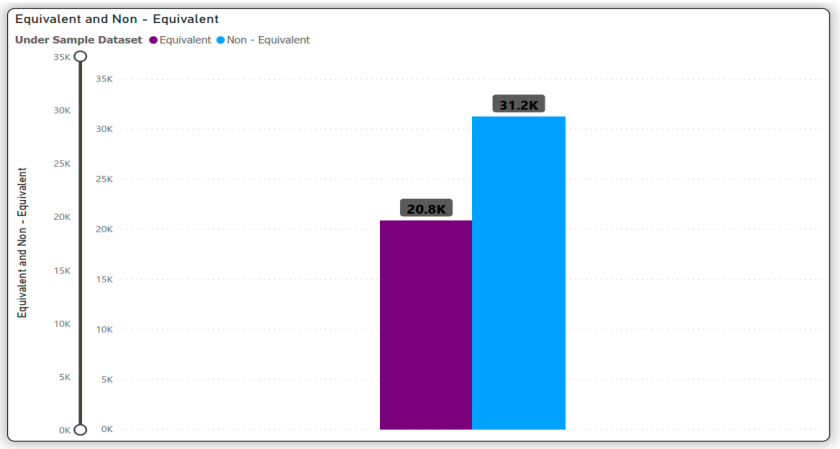

Figure 7. Our Dataset representation after Random Under Sampling.

Since the data generated was in a program language format, we used the Python library *javalang* [27] as a tool, together with *pycparser*, to parse the code into AST. It was very important to then convert the ASTs into vector representation as our model can only be fed with the vectors, as earlier explained. With the help of the embeddings, this encoding was completed before the actual training started. For the shuffling and splitting stage, the data was placed into 3 sets, *training*, *test*, and *development*, before generating the blocks for the split data set of training and testing. We constructed our embeddings using the training dataset concurrently with the unsupervised trained. Lastly, we ran the final processing of the data for the training proper, and we called this entire process the *pipelining data processing stage*.

We followed our data processing design just as we explained earlier, before feeding our TBCNN with the blocks of training, validation, and testing. We adopted this model for this research because of how effective it was in [7,23,26] and how they all had a similar problem area, program source code, although with different objectives. The model, which we implemented with kernels, paddings, and slides aside the various layers, performed well with an accuracy of 94% within 10 epochs, as depicted in Figure 8.

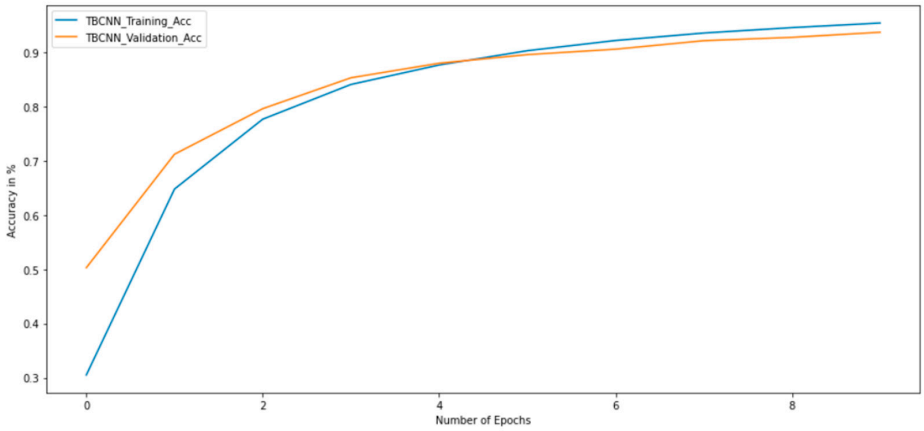

Figure 8. Training and Validation Accuracy.

As shown in Figure 8, the validation and training accuracies were around 92% and 94%, respectively. The loss function of both training and validation is shown in Figure 9. The obtained accuracy of 94% outperforms the models proposed in [2,5,7]. Furthermore, our results show fewer False Negatives during testing and lower False Positives, which is significant as it implies that there is a decrease in the risk of labeling a mutant as equivalent when it is not. This indicates a great measure in strengthening mutation testing test suites.

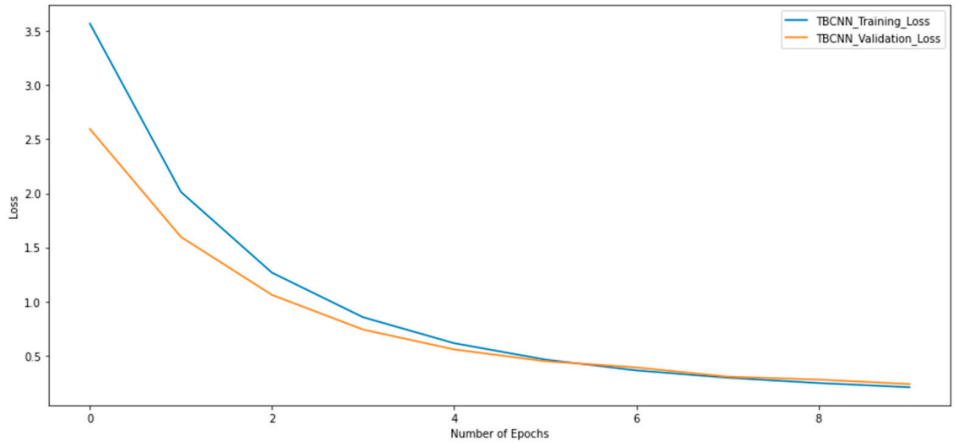

Figure 9. Training and Validation Loss.

Furthermore, to adequately evaluate our model, we achieved a 94% accuracy, higher than other models [2,5,17], which shows it predicted very few False Negatives during testing, see Figure 10. There were equally lower False Positives, thereby indicating a decrease in the risk of labelling a mutant as equivalent to the strengthening of the test suites.

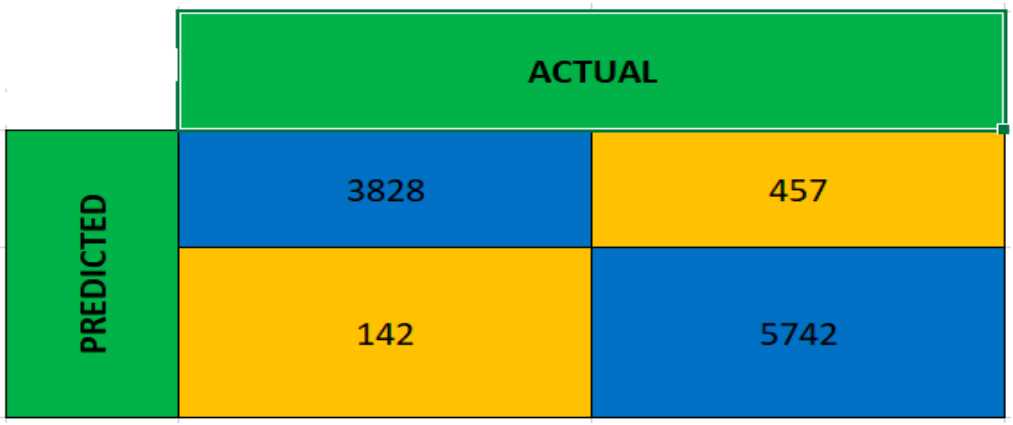

Figure 10. Confusion Matrix.

The Table 2, summarizes the result of our comparative analysis, and our standard dataset and the proposed model architecture performed very well in terms of accuracy and time taken to return results.

Table 2. Model Comparison.

Model	Accuracy	Recall	Precision	Time (s)
Binary Classification + Random Forest	80%	96%	89%	-
k-Nearest Neighbor	85%	-	-	-
ASTNN	90%	96%	100%	84.5
ASTNN (With our Standard Dataset)	94%	93%	92%	314.64
TBCNN	94%	96%	93%	153.00

The Classification report, as shown in Table 3, is a report of accuracy, recall, precision, F1-score, and total time, as provided by the performance evaluation metrics of our implantation. The resultsindicate a huge performance of the proposed model during testing. We used a dataset that was not part of the training and validation process and evaluated it with four different metrics. F1-score and accuracy showed good results when applied to balanced datasets (50–50) and produced misleading results when applied to imbalanced cases. To further evaluate our results and make sure we obtained a good model that solves the problem at hand, we used Matthew's Correlation Coefficient, MCC.

Table 3. Performance Evaluation Metrics.

Mutation Operator	Accuracy	Recall	Precision	F1-Score	MCC Score	Total Time (s)
ABS	**0.94**	0.96	0.89	0.92	0.88	153.00

MCC and F1-score are of great interest to us here because they both indicate how good our classification turned out to be. In particular, when an MCC is close to +1, it means it has high values for all the other confusion matrix metrics. From the above table, our MCC value is very high and close to +1, which further affirms the accuracy, F1-score, precision, and recall values we generated as good and within a perfect range. It further indicates the fact that the dataset we used is not overfitting or underfitting.

Equally, from the performance evaluation metrics, we will observe that our model has an F1-Score of 94%, an indication that it will have a minimal error in classifying the

equivalent mutants with fewer False Negatives and False Positives during testing, which is significant as it implies that there is a decrease in the risk of misclassification.

5. Conclusions

Mobile applications are becoming the fulcrum of human daily affairs, and the fast growth of mobile-based solutions is making the need for software testing of these fast-released apps key to the survival of the software integrity and efficiency. This paper aims to automatically classify equivalent mutants in the mutation testing of android applications using tree-based convolutional neural networks. We went through the trajectory of explaining the efficacy of using mutation testing based on previous research, where we explained the need to adopt this technique in testing android applications. We reviewed novel research papers that helped in giving us an in-depth analysis of the key issues around the equivalent mutation problem. Furthermore, to accomplish our aim, we used a tool to generate our standard dataset, transform the data into vector representation, and then generate blocks of training, development, and testing datasets. We then fed our model, TBCNN, with the trained block dataset to obtain our classification of the equivalent mutants.

At the end of the implementation, we found out that the automation of the classification of equivalent mutants is an area that needs a lot of research. We can proffer a solution to the problem using a very robust technique that we adopted: tree-based convolutional neural network (TBCNN), which shows a greater percentage of accuracy than some machine learning algorithms. Our model gave an accuracy of 94% and a validation accuracy of 92%, and the entire research further affirms the need to use a standard dataset in mutation testing. Lastly, our work will encourage the use of mutation testing in testing android applications, thereby enhancing software quality assurance.

This paper aimed to automatically classify the equivalent mutants in the mutation testing of android applications using deep learning and an abstract syntax tree, with the objective of incorporating a standardized dataset of mutants to achieve a huge reduction in the cost of finding these mutants. We adopted the MutantBench tool for generating our dataset from a collection of C and C++ SUT and, with the aid of our tree-based convolutional neural networks, achieved an accuracy of 94% and a run time of 153.00 s using GPU on Amazon SageMaker Studio Lab. We believe that this model could be utilized as a framework in the software quality industry towards improving the quality of android applications flooding the market. We recommend that more tools should be developed towards having a standardized mutant dataset, as this will increase research interest in other areas of mutation testing. More operators should be used to further implement the proposed technique, and the framework should be further developed into an android application to ease access and increase the use of the model. Lastly, other tools, such as MDroid+, MuJava, and the rest, should incorporate the standard FAIR Principle in their method of generating mutants.

Author Contributions: Conceptualization, M.B.K., B.M.-B., S.M. and I.A.S.; Methodology, S.M., M.B.K. and B.M.-B.; Data curation, M.B.K.; Formal analysis, M.B.K., B.M.-B., B.S. and I.A.S.; Writing—original draft, M.B.K.; Writing—review & editing, S.M., B.S., M.B.K. and B.M.-B. Funding acquisition and project administration, S.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The datasets and code presented in this study are available on https://github.com/mbkusharki/TBCNN-plus-AST-.git (accessed on 1 March 2022). For any other questions, please contact the corresponding author or first author of this paper.

Acknowledgments: The authors would like to acknowledge Amazon Web Services (AWS) for granting us access to use Amazon Sage maker to prepare, build, train, and test our deep learning model.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Deng, L.; Offutt, J.; Ammann, P.; Mirzaei, N. Mutation operators for testing Android apps. *Inf. Softw. Technol.* **2017**, *81*, 154–168. [CrossRef]
2. Strug, J.; Strug, B. LNCS 7641—Machine Learning Approach in Mutation Testing. In *IFIP International Conference on Testing Software and Systems*; Springer: Berlin/Heidelberg, Germany, 2012.
3. Delgado-Pérez, P.; Sánchez, A.B.; Segura, S.; Medina-Bulo, I. Performance Mutation Testing. *Softw. Test. Verif. Reliab.* **2021**, *31*, e1728. [CrossRef]
4. Salihu, I.A.; Ibrahim, R.; Ahmed, B.S.; Zamli, K.Z.; Usman, A. AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing. *IEEE Access* **2019**, *7*, 17158–17173. [CrossRef]
5. Naeem, M.R.; Lin, T.; Naeem, H.; Liu, H. A machine learning approach for classification of equivalent mutants. *J. Softw. Evol. Process* **2020**, *32*, e2238. [CrossRef]
6. Liang, H.; Sun, L.; Wang, M.; Yang, Y. Deep Learning with Customized Abstract Syntax Tree for Bug Localization. *IEEE Access* **2019**, *7*, 116309–116320. [CrossRef]
7. Peacock, S.; Deng, L.; Dehlinger, J.; Chakraborty, S. Automatic Equivalent Mutants Classification Using Abstract Syntax Tree Neural Networks. In Proceedings of the 2021 IEEE 14th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto de Galinhas, Brazil, 12–16 April 2021; pp. 13–18.
8. Escobar-Velásquez, C.; Riveros, D.; Linares-Vásquez, M. MutAPK 2.0: A tool for reducing mutation testing effort of Android apps. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, 8–13 November 2020; pp. 1611–1615.
9. Mateo, P.R.; Usaola, M.P.; Aleman, J.L.F. Validating Second-Order Mutation at System Level. *IEEE Trans. Softw. Eng.* **2012**, *39*, 570–587. [CrossRef]
10. Van Hijfte, L.; Oprescu, A. MutantBench: An Equivalent Mutant Problem Comparison Framework. In Proceedings of the 2021 IEEE 14th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto de Galinhas, Brazil, 12–16 April 2021; pp. 7–12. [CrossRef]
11. Delgado-Pérez, P.; Chicano, F. An experimental and practical study on the equivalent mutant connection: An evolutionary approach. *Inf. Softw. Technol.* **2020**, *124*, 106317. [CrossRef]
12. Kintis, M.; Papadakis, M.; Jia, Y.; Malevris, N.; Le Traon, Y.; Harman, M. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Trans. Softw. Eng.* **2017**, *44*, 308–333. [CrossRef]
13. Ma, L.; Zhang, F.; Sun, J.; Xue, M.; Li, B.; Juefei-Xu, F.; Xie, C.; Li, L.; Liu, Y.; Zhao, J.; et al. DeepMutation: Mutation Testing of Deep Learning Systems. In Proceedings of the 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE), Memphis, TN, USA, 15–18 October 2018; pp. 100–111.
14. Linares-Vásquez, M.; Bavota, G.; Tufano, M.; Moran, K.; Di Penta, M.; Vendome, C.; Bernal-Cárdenas, C.; Poshyvanyk, D. Enabling mutation testing for Android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery (ACM), Paderborn, Germany, 4–8 September 2017; pp. 233–244.
15. Moran, K.; Tufano, M.; Bernal-Cárdenas, C.; Linares-Vásquez, M.; Bavota, G.; Vendome, C.; Di Penta, M.; Poshyvanyk, D. MDroid+: A mutation testing framework for android. In Proceedings of the 40th International Conference on Software Engineering: Companion, Gothenburg, Sweden, 27 May–3 June 2018; pp. 33–36.
16. Da Silva, H.N.; Farah, P.R.; Mendonça, W.D.F.; Vergilio, S.R. Assessing Android Test Data Generation Tools via Mutation Testing. In Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing—SAST 2019, Salvador, Brazil, 23–27 September 2019; pp. 32–41.
17. Yao, X.; Harman, M.; Jia, Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May 2014–7 June 2014; pp. 919–930. [CrossRef]
18. Phan, A.V.; Chau, P.N.; Le Nguyen, M.; Bui, L.T. Automatically classifying source code using tree-based approaches. *Data Knowl. Eng.* **2018**, *114*, 12–25. [CrossRef]
19. Hu, Q.; Ma, L.; Xie, X.; Yu, B.; Liu, Y.; Zhao, J. DeepMutation++: A Mutation Testing Framework for Deep Learning Systems. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1158–1161.
20. Baer, M.; Oster, N.; Philippsen, M. MutantDistiller: Using Symbolic Execution for Automatic Detection of Equivalent Mutants and Generation of Mutant Killing Tests. In Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, 24–28 October 2020; pp. 294–303. [CrossRef]
21. Tang, D.; Qin, B.; Liu, T. Document Modeling with Gated Recurrent Neural Network for Sentiment Classification. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, Lisbon, Portugal, 17–21 September 2015; pp. 1422–1432.
22. Roy, D.; Panda, P.; Roy, K. Tree-CNN: A hierarchical Deep Convolutional Neural Network for incremental learning. *Neural Netw.* **2020**, *121*, 148–160. [CrossRef] [PubMed]
23. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing. 2014. Available online: http://arxiv.org/abs/1409.5718 (accessed on 21 March 2022).

24. Saifan, A.A.; Alzyoud, A.A. Mutation Testing to Evaluate Android Applications. *Int. J. Open Source Softw. Process.* **2020**, *11*, 23–40. [CrossRef]
25. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; Dean, J. Distributed Representations of Words and Phrases and Their Compositionality. 2013. Available online: http://arxiv.org/abs/1310.4546 (accessed on 21 March 2022).
26. Vieira, S.T.; Rosa, R.L.; Rodríguez, D.Z. A Speech Quality Classifier based on Tree-CNN Algorithm that Considers Network Degradations. *J. Commun. Softw. Syst.* **2020**, *16*, 180–187. [CrossRef]
27. Bui, N.D.Q.; Jiang, L.; Yu, Y. Cross-Language Learning for Program Classification Using Bilateral Tree-Based Convolutional Neural Networks. 2017. Available online: http://arxiv.org/abs/1710.06159 (accessed on 21 March 2022).
28. Cai, Z.; Lu, L.; Qiu, S. An Abstract Syntax Tree Encoding Method for Cross-Project Defect Prediction. *IEEE Access* **2019**, *7*, 170844–170853. [CrossRef]
29. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. 2014. Available online: https://arxiv.org/abs/1412.6980 (accessed on 21 March 2022).
30. Chicco, D.; Jurman, G. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genom.* **2020**, *21*, 6. [CrossRef] [PubMed]
31. Chicco, D.; Tötsch, N.; Jurman, G. The Matthews correlation coefficient (MCC) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation. *BioData Min.* **2021**, *14*, 1–22. [CrossRef] [PubMed]