Improving Defect Detection Using Classification on Abstract Syntax Trees

Adrian Machado

ABSTRACT

This study attempts to determine if machine learning classification algorithms can be applied on the Abstract Syntax Trees of Java code snippets to determine if the code snippet contains a software defect. The Null Pointer Exception is chosen as the basis for defect detection as it is both prominent in several codebases and can lead to critical failures of the program. Traditional tools like FindBugs use rule based algorithms to determine the presence of defects in code, however there are many exceptions to these rules leading to several false positives. In this study, several machine learning algorithms are applied to the ASTs of collected code snippets, with two distinct representations (node count and bigram) of the ASTs being used as feature vectors. The results of this analysis shows that the Bigram representation of the ASTs in conjunction with the Multilayer Perceptron algorithm is able to obtain a high recall and precision in determining if a code snippet marked as containing a Null Pointer Exception actually contains one.

1 Introduction

The absence of defects or 'bugs' is ideal in any computer software and many practices are commonly put in place during the development stage including code reviews[1] and Test Driven Development[2], to avoid defects. Despite these efforts, defects still find their way into many software products due to oversight or negligence, leading to critical issues in the software. As a result, there has been an increased focus on automated defect prediction[3] in order to find the defects that programmers miss. Many tools that perform defect prediction (known as static analyzers) like FindBugs use a rule based algorithm, looking for established patterns in code that will likely lead to a defect. There are limitations to rule-based algorithms however, as they typically flag many safe blocks of code as defects, otherwise known as false positives.

This study attempts to build on top of the rule based algorithm with machine and deep learning algorithms that will develop their own understanding of what a true defect is based on provided training data. A significant amount of training data would be required to create an analyzer that was able to predict all bugs, so the scope was limited to just the Null Pointer Exception (NPE) defect. NPEs are a common defect programmers unintentionally cause and are typically quite dangerous as they can cause a program to crash, making them a suitable candidate for testing. Figure 1 displays a typical example of an NPE, in which the program attempts to access a property of a null object.

```
public void checkObject () {
    Object o = null;
    ...
    accessProperty(o);
}

public void accessProperty (Object o) {
    Property p = o.property;
}
```

Figure 1. An NPE will occur when the program attempts to access o.property

As a developer building the accessProperty function, it may not be obvious to place a nullcheck (if statement checking if o is null) in the function. Likewise, as a separate developer building the checkObject function, initializing o may have been forgotten in-between its property access and declaration depending on the complexity of the function, thus leading to an NPE in the project. In order to determine that there is indeed an NPE in this block of code, this study proposes the use of the program's Abstract Syntax Tree (AST) as the basis of representing the code block. Furthermore, the study converts the AST into two distinct representations (node count and bigram) that attempt to summarize the AST as a numeric vector known as a Feature Vector. These feature vectors are then consumed by the three classification algorithms used (Naïve Bayes, J48 Decision Tree,

and Multilayer Perceptron) in order to classify a block of code as either buggy (containing an NPE) or clean (doesn't contain an NPE).

2 Motivation

Although static-analysis tools are capable of detecting many errors in software projects, they introduce their own set of challenges through the false positives they report.

2.1 FindBugs & False Positives

In many static analysis tools, there is always a risk of encountering false positives, and FindBugs[4] is no exception. All of the defects collected for this study are obtained using FindBugs as it is a convenient way to obtain possible NPEs from software projects. Findbugs links the source code to the analyzed jar files to display the code blocks supposedly afflicted with a bug, allowing users to manually determine if the bug is a true or false positive. In 14 out of the 18 projects analyzed, there are false positives present with regards to defects pertaining to NPEs. In many of those 14 projects, false positives make up the majority of detected bugs, which hinders the usefulness of the tool. It is therefore optimal to try and reduce the number of false positives reported. Simply switching to a different static analyzer is not a viable solution as many of them are still susceptible to similar issues[5]. The initial results provided by FindBugs should instead be analyzed in order to test for their validity, with the detected false positives being removed from the results. This study attempts to perform such analysis using machine and deep learning algorithms to identify the false positives.

3 Methods

3.1 Manually Identifying Bugs

In order to build a system that can verify the results of FindBugs or any static analysis tool, a sufficient amount of results must be collected in order to find patterns or trends in the analyzed code which the analyzer consistently makes mistakes classifying. To prevent overfitting to a specific project or publisher, I analyzed 19 different projects, many from unique organizations, using FindBugs. Only bugs labeled as 'Null Pointer Dereference' (NPD, equivalent to an NPE for our purposes) are considered for analysis. Table 1 provides a list of the projects analyzed and how many NPD bugs were found in each.

Project Name	Number of NPDs
Flume	3
Elasticsearch	5
Helix	5
Camel	4
Tomcat	3
Truth	2
Guava	1
Mssql-jdbc	1
Cglib	2
Mockito	1
Giraph	2
Guice	3
Jnr-posix	2
Sanselan	5
Casmi	3
Esri Spatial Framework	2
Ovea.jetty-session-redis	1
Kornakapi	1
Scale7-core	1
Total	45

Table 1. Projects analyzed using Findbugs and the number of Null Pointer Dereferences found in each

Upon collecting the results from FindBugs, I inspect each of the bugs manually, looking at the snippets of code that are flagged by FindBugs. For each NPD, I read the warning message provided by FindBugs to determine its justification for

labeling the snippet as buggy, and through testing of the method containing the NPD, and inspection of the project's codebase, I can determine whether the NPD is a true or false positive. This process needs to be rigorous as the labeling on the training data is based on this manual classification. An example of a true and false NPD can be seen in Figure 2 and Figure 3 respectively.

```
public void trueNPE (Boolean flag) {
        NativeObject a = null;
        if (flag == true) {
                a = new NativeObject();
        }
        int len = a.length;
}
```

Figure 2. A true positive, a won't be set if flag is false, leading to an NPE when a.length is accessed

```
public void falseNPE(Boolean flag) {
        NativeObject a = null;
        if (flag == true) {
                a = new NativeObject();
        } else {
        return;
        }
        int len = a.length;
}
```

Figure 3. A false positive, a.length will never be accessed if it is not initialized, findbugs can't determine this however and marks it as a bug

3.2 Building Feature Vectors

3.2.1 Creating ASTs

Once all of the NPDs are sorted into true and false positives, their respective code snippets must be converted into a general form in which they all share the same attributes. This is done so a generic representation of any code snippet can be achieved for analysis purposes. In Java, program can be represented using its Abstract Syntax Tree (AST), a representation of the structure and actions of the program in terms of 'nodes', individual statements the compiler can interpret. An AST essentially removes all of the project specific details of the code, for example, a method declaration with some custom object as the return type, and custom classes as the parameters is simple represented as a 'MethodDeclaration' node in the AST[6]. A source code to AST compiler is built using Eclipse JDT to automate the conversion process. Rather than import and build every single project, I create generic Java objects and methods to replace any project specific objects and methods in the source code snippets (ex. DummyObject.getString() replaces all method calls returning strings), as only the nodes need to remain unchanged to produce the same AST, while the functionality is arbitrary.

Due the verboseness of Java, every code snippet typically contains many AST nodes, many of which are the result of boilerplate code, rather than contributing towards the algorithm in the snippet. I choose to have my compiler only list the following AST nodes; MethodDeclaration, MethodInvocation, FieldDeclaration, VariableDeclaration, ClassInstanceCreation, FieldAccess, IfStatement, ReturnStatement, ForStatement, WhileStatement, InfixExpression, ConditionalExpression, TryStatement, CatchClause, ThrowStatment, NullCheck, NotNullCheck. NullCheck and NotNullCheck are actually custom AST nodes built for this study that represent equivalence checks (ex. x == null) and non-equivalence checks (x != null) with regards to null. The reason these were created is that the null primitive is not uniquely identified by any standard AST node, but nullchecks and notnullchecks are a common pattern in code that is flagged as containing an NPE therefore I hypothesize that they may provide insight as to whether the NPD is a true or false positive. Once the code snippets are all converted to ASTs, I then manually inspect each of the ASTs to determine if they are eligible for analysis depending on the number of nodes present and reliance. If an AST only contains 3 or fewer nodes, it typically does not provide enough information as to whether it is a true or false positive as it is too short to represent a meaningful pattern. Upon removing the ineligible NPDs the following results remain.

The remaining ASTs all contain nodes from the pre-selected set however these nodes are still ordered based on their order appearance in the program. This ordering is essentially random and not easily interpretable by conventional machine learning

Project Name	Number of NPDs	True Positives	False Positives
Flume	3	1	2
Elasticsearch	5	4	1
Helix	5	3	2
Camel	4	3	1
Tomcat	3	1	2
Mssql-jdbc	1	0	1
Cglib	2	1	1
Mockito	1	0	1
Giraph	2	1	1
Guice	3	1	2
Sanselan	5	0	4
Casmi	3	0	3
Esri Spatial Framework	2	0	2
Ovea.jetty-session-redis	1	1	0
Kornakapi	1	0	1
Scale7-core	1	1	0
Total	41	17	24

Table 2. The remaining NPDs after filtration based on their ASTs

classifiers so a new representation of these nodes is required. This representation is called a Feature Vector as it contains information about some data (features) and is represented in terms of a numeric vector (ex. [1, 3, 4]).

3.2.2 Node Count Representation

One way to represent an AST in terms of its nodes while adhering to a specific format is by counting the number of times a node shows up in an AST (each count is a feature) and recording that count at a preselected index in a feature vector. For every feature vector, each index corresponds to one of the nodes in the chosen set, and same indexes correspond to the same nodes between two feature vectors. For example, Figure 3 contains a ReturnStatement, while Figure 2 does not, so the count at index 7 (index corresponding to ReturnStatements) would be 1 and 0 respectively in their feature vectors.

This is a simple way of representing the AST, and is based on the idea that a higher or lower count of certain nodes can indicate whether or not a code snippet contains an NPE. There are limitations to the effectiveness of this representation however, as it discards the order of the AST essentially causing the loss of context from the program. It is for this reason, another representation is required.

3.2.3 Bigram Representation

Bigram (pronounced as bi-gram) is based on the N-gram[7] format using a gram size of 2. The algorithm to build a bigram representation performs a similar operation as the Node Count representation however it preserves some of the order. This is achievable by storing the occurrence of consecutive pairs of nodes rather than just the count of individual nodes (ex. if a MethodInvocation comes after a NullCheck, the count at the index corresponding the NullCheckMethodInvocation is incremented) in an n*n matrix (where n is the number of chosen AST nodes). This representation provides information on which nodes precede/succeed each other, and depending on the data, the probability of certain nodes following each other can be determined[7] thus the reconstruction of portions of the AST is possible.

3.3 Analyzing the AST Representations

Through the use of machine and deep learning classifier algorithms, a model can be created that attempts to determine whether or not a code snippet truly contains an NPE. An extra feature is added to each feature vector, 'ValidBug', which is a Boolean value that represents whether the feature vector (and therefore its corresponding code snippet) actually contains an NPE (ValidBug = yes) or is a false positive (ValidBug = no) as determined in 3.1. This field is classified upon using Weka's suite of classification algorithms, namely the Naive Bayes, J48 Decision Tree and Multilayer Perceptron algorithms.

3.3.1 Naïve Bayes

The Naïve Bayes classifier is a probabilistic machine learning classification algorithm that uses Bayes theorem and the assumption that all features in a feature vector are independent of each other[8]. Given a feature vector $F = [f_1, f_2, \ldots, f_n]$ where $f_1, \ldots, f_n$ are features, the probability of it being classified as a valid bug is as follows

$$P(ValidBug|F) = \frac{P(F|ValidBug)P(ValidBug)}{P(F)}$$

Naïve Bayes states that F can be classified as a ValidBug if

$$\frac{P(ValidBug|F)}{P(!ValidBug|F)} >= 1$$

Due to the assumption of independence this can be simplified to

$$\frac{P(ValidBug)}{P(!ValidBug)} \sum_{i=1}^{i=n} \frac{P(f_i|ValidBug)}{P(f_i|!ValidBug)}$$

This algorithm, as suggested by its name, is not always accurate as the assumption of independence is not always true. For example, a return statement wouldn't appear without a method declaration so there is a correlation between their appearances. The main benefit of the algorithm is that it is fast so obtaining results providing preliminary insight into the data is possible, and the impacts of changes to the testing/training configurations can be quickly be assessed.

3.3.2 J48 Decision Tree

Decision tree algorithms create n-ary trees known as decision trees that consist of two types of nodes, decision nodes, and result nodes. Decision nodes look at a single feature in a feature vector and based on the feature, splits testing samples into groups based on a decision rule, represented by child nodes. These children can be result nodes, in which the node is the trees' decision as to which of the n classes that vector belongs to, or they can be decision nodes meaning the tree must assess more features before making a decision as to which class the vector belongs to[9]. Decision trees are generated using different algorithms, with J48 being generated using the C4.5 algorithm[9] however such algorithms for construction are not strictly relevant to this study. Although J48 is slower than Naïve Bayes, it typically produces a model with more accurate results[9].

3.3.3 Multilayer Perceptron

The Multilayer Perceptron algorithm is a deep learning algorithm that makes use of a neural network for classification using backpropagation. Simply put, the neural network consists of several nodes, some on the input layer that receive the feature vectors, many in the 'hidden' layers that process the feature vectors, and the remaining on the output layer which returns the classification results[10]. Neural networks take a significant amount of time to build and train, however they make use of several layers of computation, typically leading to very accurate results.

4 Results

4.1 Evaluation Metrics

Before interpreting the results procured by Weka, a set of evaluation metrics is required in order to compare the three algorithms. With the objective of reducing the number of false positives in mind, it naturally follows that metrics involving the ratio of true positive count to false positive count would be a good indicator of improvement, hence the use of precision is ideal. The formula of precision is as follows:

$$\frac{TP}{TP+FP}$$

Where TP is the true positive count, the number of code snippets correctly identified by the classifier as containing an NPE, while FP is false positive count, the number of code snippets incorrectly identified by the classifier as containing an NPE. It is also important to understand the context of the problem however, as underreporting or missing defects makes the tool less useful than one that has more false positives but reports all of the true positives. Recall is therefore used as the second evaluation metric, with the formula as follows:

$$\frac{TP}{TP+FN}$$

Where FN is the false negative count, the number of code snippets incorrectly identified by the classifier as not containing an NPE.

4.2 Evaluation Results

A complete set of results can be found in table 3. In each test, 10 fold cross-validation is used to compute the results. For comparison, the classification initially provided by Findbugs had 17 true positives, and 24 false positives, resulting in a precision of 0.414.

AST Representation	Classification Algorithm	# of TP	# of TN	# of FP	# of FN	Precision	Recall
Node Count	Naïve Bayes	6	19	5	11	0.545	0.353
Node Count	J48	8	17	7	9	0.533	0.471
Node Count	Multilayer Perceptron	12	16	8	5	0.6	0.706
Bigram	Naïve Bayes	10	16	8	7	0.556	0.588
Bigram	J48	7	19	5	10	0.583	0.412
Bigram	Multilayer Perceptron	14	14	10	3	0.583	0.824

Table 3. Results from testing different AST representation and classification algorithm combinations

4.2.1 Evaluating Naïve Bayes

As predicted, the Naïve Bayes Algorithm trained using the Node Count representation created the least performant model, with a passable precision score, but low recall score. This low recall score effectively makes the model unusable for bug reporting for the reasons stated in 4.1. The usage of a Bigram representation resulted in a sizeable increase of 0.235 in recall score and a marginal increase in precision, lending credence to idea that partially preserving the ordering of nodes helps determine whether or not the code snippet contains an NPE.

4.2.2 Evaluating J48 Decision Tree

Also as predicted, the J48 decision tree produced a model superior to the Naïve Bayes using the node count representation, with a significantly higher recall, and only marginally lower precision score. This is not true for the Bigram representation however, as a noticeable drop in recall (0.059) was recorded, contrary to the boost in recall score observed in the Naïve Bayes model. This drop in recall is countered by a 0.05 jump in precision score. The low recall score again makes this model ineffective as an NPE detector, with the majority of bugs being labeled as false negatives.

4.2.3 Evaluating Multilayer Perceptron

The Multilayer Perceptron algorithm produced the best results overall for both the Node Count and Bigram representations, with the Node Count model having a precision of 0.6 and a recall of 0.706. Both of these values are significantly higher than the maximum values of the J48 and Naïve Bayes models. The Bigram model introduces a tradeoff between precision and recall, with a small drop in precision (down to 0.583, same as the J48 model) but a large increase of 0.118 in recall. This model is especially promising as it essentially preserve the vast majority of true bugs, while filtering out more than half of the false positives. Given the original precision of Findbugs being 0.414, a precision of 0.583 represents a 40.8% gain in precision.

5 Threats to Validity

5.1 Overfitting

The major threat to the validity of the results comes not from the training of the models, but from the data they are being trained on. Only 41 instances of null pointer defects were collected in the data collection phase of this experiment, and out of those results, only 17 were actually true positives. Ideally, the models would be trained on hundreds of instances from dozens of projects so the model is not overfitted to the small set of data provided to it. This is apparent in the Bigram J48 decision tree, as it only has to consider a small set of AST nodes in order to perform classification (Figure 4), but training with more projects and data samples would most likely result in a more complex tree (similar to the Node Count tree in Figure 5), and quite different results. Collecting and training with more instances of possibly buggy code will be needed to mitigate this threat.

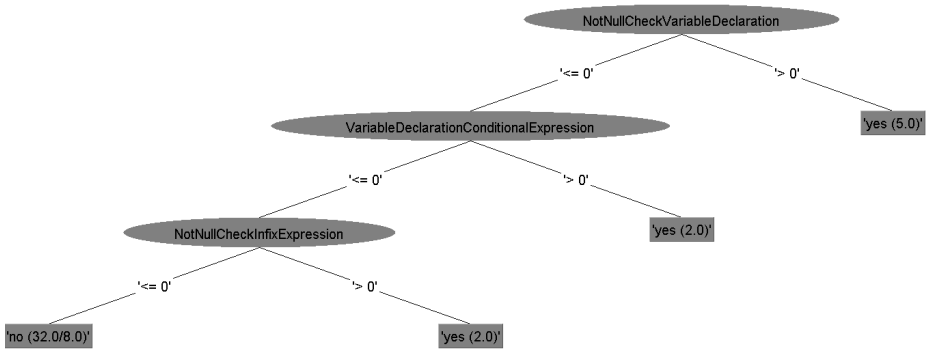

Figure 4. Decision tree produced using the Bigram representation

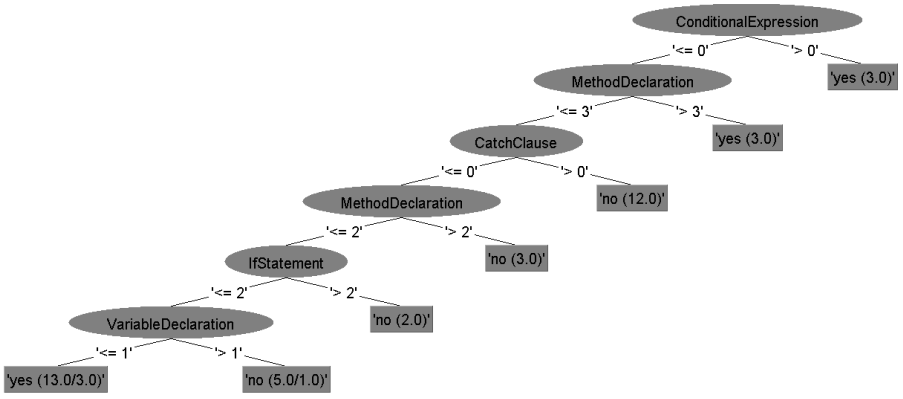

Figure 5. Decision tree produced using the node count representation

6 Conclusion

This paper proposes the usage of a Multilayer Perceptron model trained on a Bigram representation of the AST's of possibly buggy source code in order to improve the results of traditional, rule-based algorithms for finding bugs in source code. The specific metrics that measure improvement are the precision and recall of the model, with regards to identifying Null Pointer Exceptions in source code. The model improves the precision score by 40% while retaining a high recall score of 0.824, suggesting that the Multilayer Perceptron in conjunction with the Bigram representation can effectively filter out false positives from the results traditional bug-finding tools, while not significantly increasing the number of true negatives.

References

1. McIntosh, K. Y. A. B. . H. A. E., S. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. *Proc. 11th Work. Conf. on Min. Softw. Repos.* 192–201 (2014).
2. Janzen, . S. H., D. Test-driven development concepts, taxonomy, and future direction. *Comput.* **38**, 43–50 (2005).
3. Wang, L. T. . T. L., S. Automatically learning semantic features for defect prediction. *Proc. 38th Int. Conf. on Softw. Eng.* 297–308 (2016).
4. Ayewah, . P. W., N. The google findbugs fixit. *Proc. 19th international symposium on Softw. testing analysis* 241–252 (2010).

5. Johnson, S. Y. M.-H. E. . B. R., B. Why don't software developers use static analysis tools to find bugs? *ICSE* **13** (2013).

6. Fischer, L. J. . v. G. J. W., G. Abstract syntax trees-and their role in model driven software development. *Softw. Eng. Adv.* 38–38 (2007).

7. Wang, C. D. M.-A. D. . T. L., S. Bugram: bug detection with n-gram language models. *Proc. 31st IEEE/ACM Int. Conf. on Autom. Softw. Eng.* 708–719 (2016).

8. Zhang, H. The optimality of naive bayes. *AA* (2004).

9. Patil, . S.-S. S., T. R. Performance analysis of naive bayes and j48 classification algorithm for data classification. *Int. J. Comput. Sci. Appl.* **6**, 256–261 (2013).

10. Ruck, R. S. K. . K. M., D. W. Feature selection using a multilayer perceptron. *J. Neural Netw. Comput.* **2**, 40–48 (1990).