Towards a Machine Learning Approach for the Detection of Null Pointer Dereference Errors

Charlie Molony
Supervisor: Vasileios Koutavas

April 17, 2023

Acknowledgments

I would first like to thank my supervisor, Professor Vasileios Koutavas for his continued support and guidance throughout this project. His advice was incredibly helpful.
I would also like to thank Dr Francois Pitie for meeting with me and clarifying some technical concepts.
I would like to thank my parents, Jack and Deirdre, and my brother Tom for their incredible support and encouragement throughout this project. Lastly, I would like to thank Paraic and Conor for their support throughout this process.

Abstract

Null pointer dereferences, (NPD), are common software defects that occur in a number of programming languages. The programming language C was investigated in this project. NPDs have been known to cause application crashes and system failures. NPDs are challenging to detect due to their dynamic nature. Current software tools do not provide a complete remedy for detecting null pointer dereferences. This project aims to investigate a machine-learning approach for the detection of NPDs in software programs.

A database of code containing NPDs was created. This text data will be used to develop models. A variety of machine learning algorithms were investigated and their performances were evaluated.
A multilayer perceptron model produced had relatively performant results. Transformer models did not perform well in comparison to the alternative models investigated. Models' performances were hindered by irrelevant text data, which did not contribute to the existence of NPDs.
This project did not aim to produce a perfectly operational model but to serve as a stepping stone for the use of machine learning in this area. The findings in this project may be built upon in future to create tools that perform better in the detection of NPDs. The lessons learned may also be applied to a machine learning approach to remedy other software defects, not specifically null pointer dereferences.

Contents

1 Introduction

1.1 Null Pointer Deference

In 2009 Sir Tony Hoare referred to null pointer references as his ”Billion dollar mistake” [1].Null Pointer Dereferences occur when an application dereferences a pointer variable that has a null value. This typically results in a crash or exit. Here, a null pointer is used as if it were pointed at a valid memory area. Since 2019, NPDs have been a part of “CWE Top 25 Most Dangerous Software Weaknesses.” NPDs have the potential to crash a process to cause a denial of service. They can also be used to execute an arbitrary code under specific conditions[2].

by leveraging such errors. This is done by crashing a process of denial of service.

Below is a simple program that would cause a null pointer to dereference:

```
#include <stdio.h>

int main() {
    int *ptr = NULL;
    *ptr = 3; // dereferencing NULL pointer
    return 0;
}
```

This program attempts to assign the value 3 to the memory of *ptr* using the dereference operator (*) but since *ptr* is a NULL pointer, this operation is invalid and results in an NPD.

1.2 Current methods of dealing with a null pointer dereference:

NPDs are currently remedied by formal verification, testing, runtime verification and code analysis. However, none of these have been able to provide a cure-all solution. Formal verification is a technique that leverages mathematical methods to prove the correctness of algorithms [3]. It can be used to check the behaviour of the system, finding NPDs in the process. Runtime verification is an execution approach based on extracting information from a running system [4]. This can be used to find NPDs by specification-based monitoring, dynamic analysis, and symbolic execution [5].

1.3 Investigation into machine learning:

Despite significant efforts to minimise these errors they still persist in software development. As such, a solution may be found through the application of machine learning (ML) techniques. These are effective at learning patterns that might go overlooked by a human. The hypothesis of this project states that programs with NPDs contain textual information in their code that would allow an ML algorithm to recognise them. Furthermore, the results of this project could lead to the creation and development of more effective tools for detecting and preventing NPDs. An ML model would approach this task in a very different way than traditional verification tools, leading to very different outcomes. This discrepancy could leverage the strengths of both methods of detecting NPDs, improving the overall reliability and security of software systems.
There have already been investigations into fixing null pointer exceptions in the programming language JAVA, but there are no known investigations into specifically detecting NPDs in C code.
Only supervised learning will be investigated in this project with a traditional ML and a transformer approach compared. The traditional ML approach will involve building models using algorithms such as support vector machines (SVMs), extreme gradient boosting (XGboost), and neural networks. The transformer approach will retrain one of Microsoft's Codebert neural networks. There were no known datasets containing labelled code snippets, therefore, a dataset needed to be created. This was accomplished through the use of GitHub commits and their corresponding commit message.

1.4 Specifications:

All code in this project was written in Python and run on a Dell Inspiron 14 with an i5 Intel processor. Select computationally-intensive models were trained on a virtual machine hosted by Trinity College Dublin.

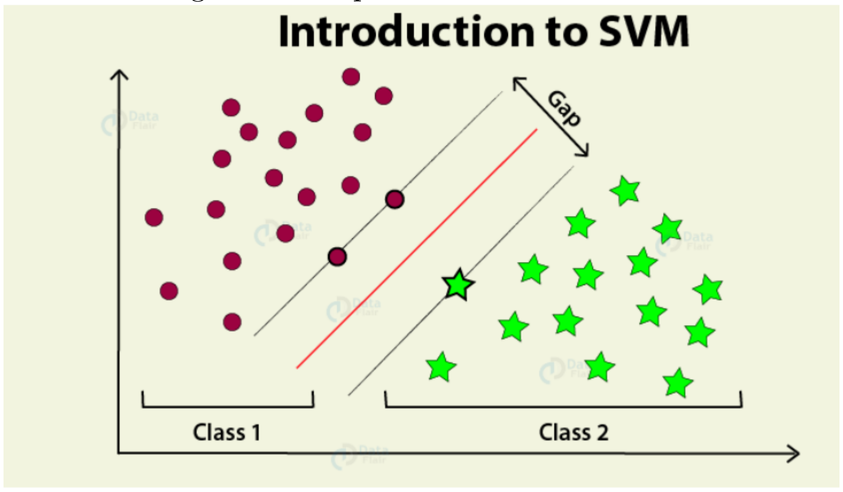
2 Relevant Materials

2.1 Core Machine Learning Concepts

2.1.1 Support Vector Machine

SVM was once the most popular classification technique. SVM is based on linear classifier $y = [x^T w > 0]$, a linear classifier is a model that makes its predictions based on a linear combination of its input variables[6].
The loss of function of SVM is based on the Hinge loss function:
$L_S VM(w) = \sum_{i=1}^{N}[y_i = 0]max(0, x_i^T w) + [y_i = 1]max(0, 1 - 0, x_i^T w))$
SVM tries to find the hyperplane that maximises the separation between two classes [7].

Figure 2.1: Separation of Two Classes

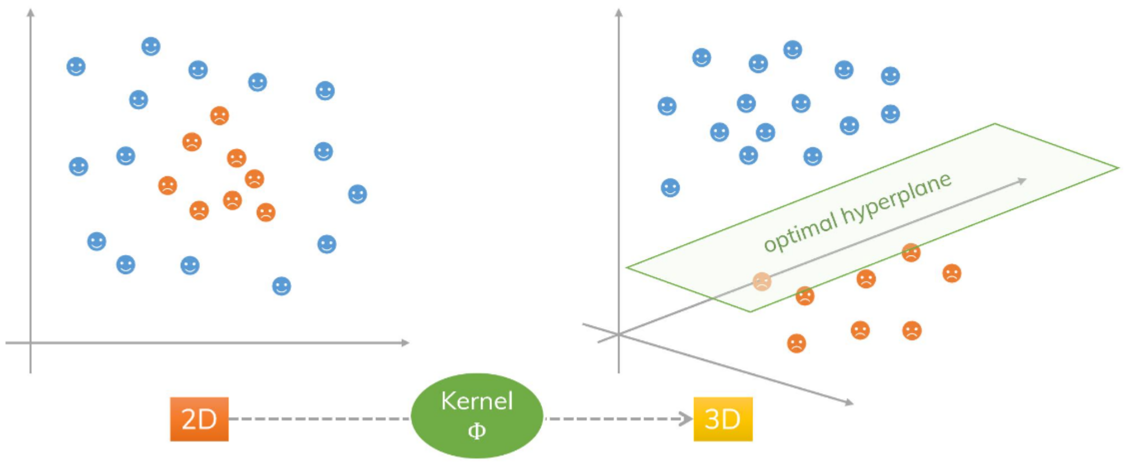


2.1.1.1 Kernel Trick

The kernel trick is a technique that can be applied to SVM. The kernel trick maps low-dimensionality data to a higher dimensional space. This gives SVM non-linearity. The original features are mapped into a higher dimensionality feature space [8].As data gets more complex the dimensionality of the features increases. This also increases the complexity of training.
.

$$\phi(\mathrm{x})=\begin{pmatrix}1\\x\\x^2\\x^3\\\dots\end{pmatrix}$$

Figure 2.2: Kernel Trick



2.1.2 XGBoost

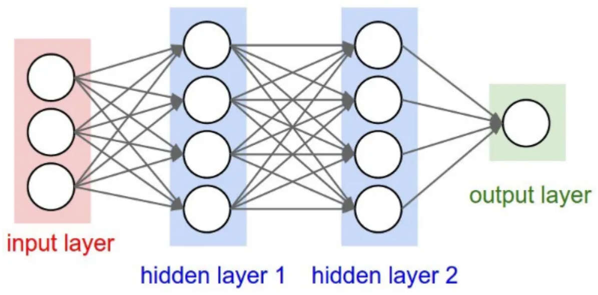
XGBoost is based on the decision tree algorithm. Decision trees make predictions based on how a previous set of queries were answered [9].XGBoost gets its name from extreme gradient boosting which iteratively adds decision trees to improve accuracy. It is an ML algorithm that has gained a large amount of attention in the last few years due to its excellent performance in a variety of tasks like classification, ranking and regression. It is based on the traditional gradient-boosting algorithm and is favoured for its ability to handle large datasets. It can also be easily customised using its range of hyperparameters. One study found that XGBoost outperforms other algorithms in accuracy and speed in binary classification tasks [10]. Another study found that XGBoost achieved state-of-the-art results for human activity recognition[11].

2.1.3 Feed Forward Neural Networks

Neural networks are a very popular and powerful type of ML algorithm. Neural networks are employed due to their ability to solve complex problems, having produced state-of-the-art ML models capable of solving complex problems. Recently, they have been the most performant algorithm in regard to natural language tasks[12].
Neural networks are organized layers of nodes. The nodes are a non-linear function of the output of the nodes in the layer preceding it and the weights assigned to each connection

of these nodes. Each node has a threshold that must be met before it can activate. The nonlinearity of the activation functions allows neural networks to solve complicated problems. Deep neural networks may learn hierarchical data representations because of their numerous layers of neurons. [13].

Figure 2.3: Feed Forward Neural Network Structure



2.1.3.1 Training

At its core, neural nets evaluate a non-linear activation function f based on the input $\mathbf{x} = (x_1, x_2, ..., x_n)$ and weights $\mathbf{w} = (w_1, w_2, ..., x_m)$ and return the output values $\mathbf{y} = (y_1, y_2, ..., y_r)$:

$$f(x_1, .., x_n, w_1, ..., w_m) = (y_1, ..., y_r)$$

Training of neural nets is based on optimising the weights that would best solve the problem at hand. The loss function is a comparison of the predicted result to the observed result. The loss function is minimised. This is done by computing the gradient of the loss function with respect to its weights. The backpropagation algorithm is used to do this. The gradient descent approach uses the computed gradient of the loss function with respect to its weights to find the local minima [14].

The Gradient Descent Function: $w_{n+1} = w_n - \eta \frac{\partial E(w)}{\partial w}(w_n)$.
E(w) is the loss function, w_n is the value of the weight before gradient descent is computed, w_{n+1} is the value of the weight once computed. η is the learning rate, this controls the speed of descent. Backpropagation is used to compute $\frac{\partial E(w)}{\partial w}$.

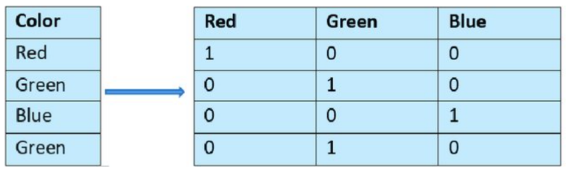
2.1.4 Word Embeddings

In natural language processing, word embedding is the representation of a word. The representation is a real-valued vector that tries to encode the meaning of a word so that words closer in the vector space have a similar meaning[15].

2.1.4.1 One Hot Encoding

This is the most simple example of representing words as a vector. A vector is created that is the size of the total unique words. Each word pertaining to a vector's index is given a value of 1, and the other words are given a value of 0[16].

Figure 2.4: One Hot Encoding



Color
Red
Green
Blue
Green

Red	Green	Blue
1	0	0
0	1	0
0	0	1
0	1	0

The advantages of one hot encoding are that it is binary and the results sit in orthogonal vector space. The disadvantages are that it produces high-dimensionality feature space that does not retain word meaning.

2.1.4.2 Term Frequency-Inverse Document Frequency

This statistical measure determines the significance and importance of specific words in documents. Term frequency works by finding how often a particular word occurs in a document. Inverse document frequency refers to how common a word appears in its corpus. Values for TF and IDF are found using different equations and then the product of these terms results in the value for the words vector[17].
In the context of this project, TF-IDF may produce results that are performant, however as TF-IDF does not retain word location in a given text dataset it is impossible to produce a model that will understand the complex semantic relationship between word location and its effect on the code. For example:

Example 1:

```
#include <stdio.h>

int main() {
    int  Example = 10;
    int * num =NULL;
    int * ptr = num ;
     * ptr = Example; // &

    return 0;
}
```

Example 2:

```
#include <stdio.h>

int main() {
    int * Example = NULL;
    int num = 10;
    int * ptr = & num; //  *
    ptr= Example;

    return 0;
}
```

Example 3:

```
Example ; ; ; ; ; ; Example

return
main()
0

} <stdio.h>
num
num
& * * *
//
 int int int int  #include
{
```

As can be seen from the toy examples. Example one would cause a null pointer to dereference due to *ptr* being assigned to the value of the pointer of *num* which is *NULL*, this is then dereferenced to a valid memory address. Therefore this causes an NPD. Yet example 2 would not cause an NPD as *ptr* is assigned the memory address of *num*, only after that is it assigned the value of *Example* which is *NULL*. No null pointers were dereferenced in example 2. Example 3 is an invalid piece of code that is completely nonsensical. All three of these code snippets would appear as exactly equal to the machine learning model if the words were vectorized using the TD-IDF approach as they have the exact same vocabulary. Therefore in this case this may not be the most valid approach for this project.
However, it may produce very performant results as *if not NULL* statements often avoid null pointer dereferences it could be possible to produce a model that checks for the existence of *if not nul* statements. It is important to note that TF-IDF would count *int** as 1 word and *int ** as 2 separate words even though in a C program they are exactly equal.

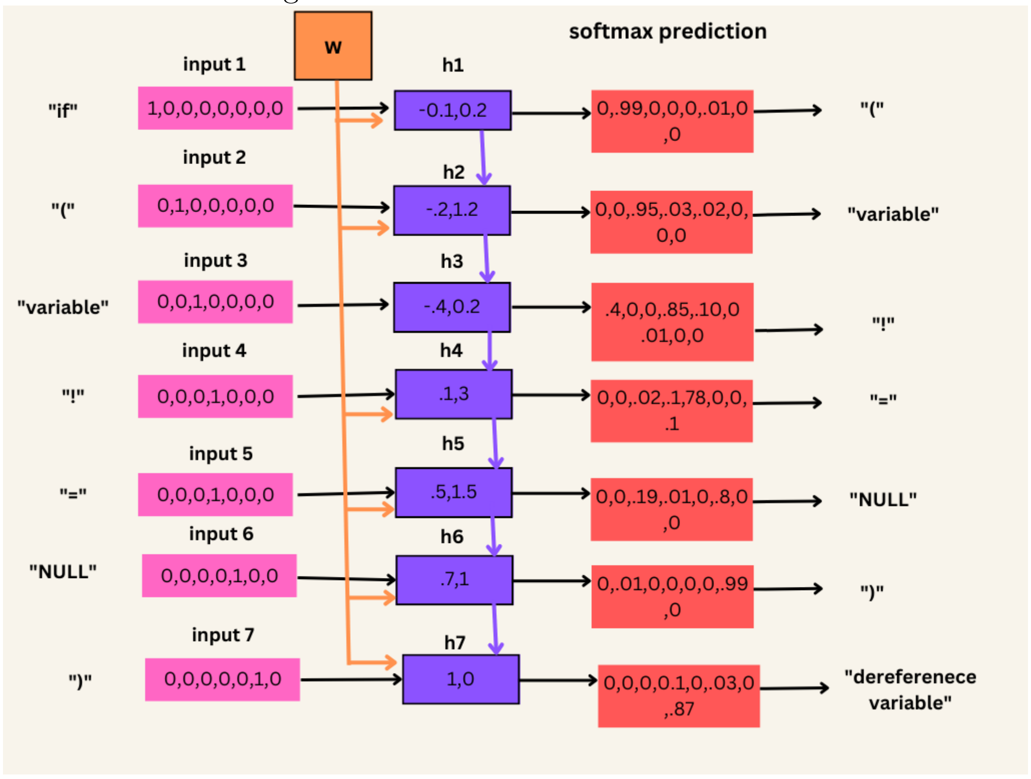
2.1.5 Transfer Learning

Transfer learning is the process of using a pre-trained neural network for different but related tasks [18]. For example, a neural network that had been trained to detect monkeys could be used and retrained to produce a model that detects chimpanzees. Transfer learning usually entails using a model that is more general and has been trained on lots of data being retrained on less data for a very specific task. Transfer learning is popular because when two models share their knowledge they can produce a more powerful and accurate model. It is a very common tool used in NLP as it lends itself to using less data for training as the model has already been trained on a large dataset, learning the basic features of the language. It is not possible to create a dataset with millions of code snippets of NPDs. Therefore using transfer learning may be essential for the production of an operational model.

2.1.6 Recurrent Neural Networks

Recurrent Neural Networks are a type of neural network that is used for sequential data. RNNs define a recursive evaluation of a function. The input stream feeds a layer called the context layer, which then re-uses the previously computed values to compute the output values. It is important to note that weights in RNN are shared across iterations [19].
The diagram below shows how an RNN would fix a null pointer dereference error. The diagram represents is a theoretical example that uses one-hot encoding to represent the words in the code as a vector.

Figure 2.5: Recurrent Neural Network



In simple terms, RNNs re-use the output of one of the preceding layers as the input to the next layer. As can be seen from the diagram, the RNN predicts that after *'if'* the RNN predicts that *'('* will be the next 'word', this is then the input to the next iteration. The output of this RNN could fix an NPD:

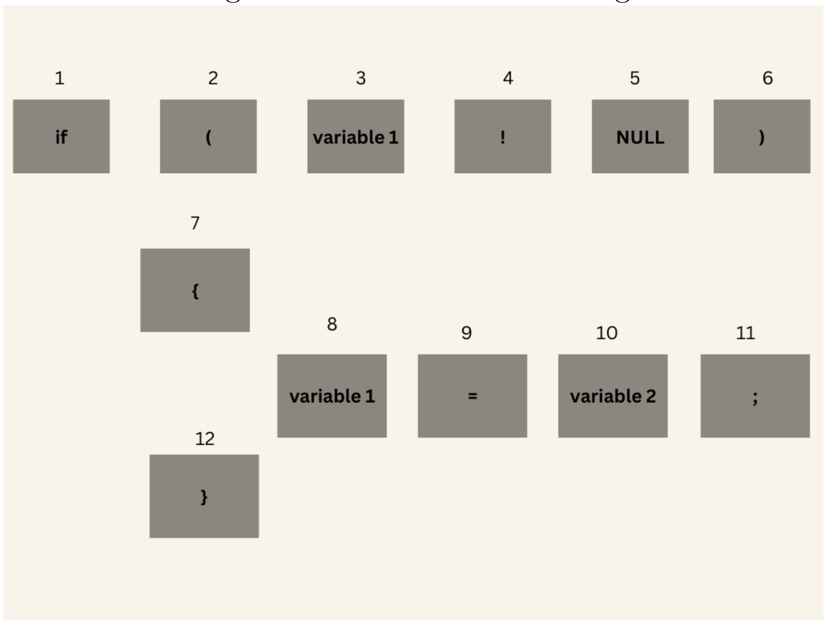
```
if ( variable ! = NULL) dereference variable
```

RNNs suffer from the problem of exploding and vanishing gradients [20].Another big problem of RNNs is that training cannot be parallelised as sequence order is part of the structure of the network [21].For these reasons, RNNs are not a practical network structure as they cannot analyse large amounts of text data. They do however provide the building blocks for the transformer model.

2.1.7 Transformer Models

Transformer models have revolutionised the field of natural language processing. Transformers enable the training of deep neural networks on large amounts of text data. Transformers use positional encoding to store information about word order in the data itself, rather than the structure of the network. Once the network has been trained on lots of text data, it learns how to interpret the positional encodings, and the importance of word order from data, not how the network was structured[22].The positional data is inputted to the transformer with the text data:

Figure 2.6: Positional Encoding



Transformer models are based on an encoder-decoder architecture. The encoder's job is to step through the input time steps and encode the entire sequence to a fixed-length vector called the context vector. The decoder's job is to step through the output time step while reading from the context vector. Transformer models are built on the attention mechanism. The attention mechanism is an adaptation of deep neural networks so that it captures representative features of a given classification task. Self-attention is used in transformer models. Self-attention allows a text model to make a decision with context to all other words in the inputted text[23]. This is advantageous as it pays greater attention to specific factors of a problem. It is very beneficial for sequence-to-sequence tasks, such as the problem in this project. Unlike vectorizing with TF-IDF transformers take into account the position of the word in the text. This fact lends itself to using transformers in this project.

2.1.8 Microsoft's CodeBert

Microsofts CodeBert model is a state-of-the-art model that is developed for source code tasks [24]. This model is very popular in the machine-learning community due to its exceptional performance on various tasks. It is a transformer-based neural network that has been pre-trained on a large amount of source code. It has proven to perform code summarization, completion, documentation and of course classification.

2.1.9 Hybrid Learning

Hybrid learning is an approach that combines two machine learning algorithms to improve accuracy and overall performance. Hybrid learning has gained popularity, particularly the integration of XGBoost and neural networks. This study [25]produced performant results when using a neural network which is then followed by XGBoost. The neural network learns high-level feature representation of data. The XGboost model is then used for prediction.

2.2 Evaluation of Machine Learning Methods

It is important to have quantifiable metrics to compare all of the models produced in this project. The following metrics will all be used in the discussion to evaluate model performance. As there are two types of errors, False Positives and False Negatives, two metrics will always be needed to properly evaluate the performance of a classifier.

2.2.1 Recall/Sensitivity/True Positive Rate (TPR)

The probability that a positive example is predicted as positive:

$$Recall = \frac{TP}{P} = \frac{TP}{TP+FN} = P(\hat{y} = 1|y = 1)$$

2.2.2 Precision

The probability that a positive prediction is indeed positive:

$$Precision = \frac{TP}{TP+FP} = P(y = 1|\hat{y} = 1)$$

2.2.3 False Positive Rate (FPR)

The proportion of negatives that are incorrectly labelled as positive:

$$FPrate = \frac{FP}{N} = \frac{FP}{TN+FP} = P(\hat{y} = 1|y = 0)$$

2.2.4 Accuracy

Accuracy is the probability that a prediction is correct:

$$Accuracy = \frac{TP+TN}{P+N} = \frac{TP+TN}{TP+TN+FP+FN} = P(\hat{y} = 1|y = 1) + P(\hat{y} = 0|y = 0)$$

2.2.5 F1 Score

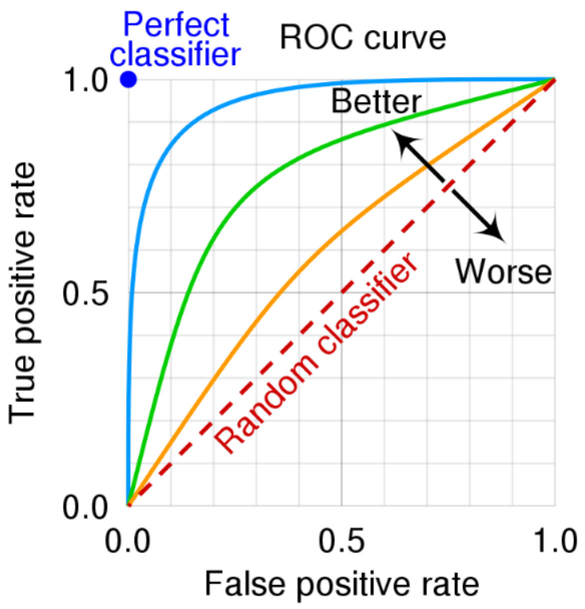
F1 score is the harmonic mean of precision and recall:

$$f_1 = 2 * \frac{recall*precision}{recall+precision} = \frac{2TP}{2TP+FP+FN}$$

2.2.6 ROC curve

When measuring the performance of a classifier it is possible to change the value of threshold value, T, which results in the classification of positive or negative. For example, an XGBoost model normally classifies a data point as positive if the threshold value is greater than 50%. Therefore by varying this threshold [26]value a family of classifiers is produced with different False positivity Rates and True Positivity Rates. These values can be used to plot a receiver operating characteristic curve.

Figure 2.7: ROC Curve



2.2.6.1 ROC-AUC

AUC stands for "Area under the ROC curve". This is the area under the curve. AUC provides a measure of performance overall classification thresholds[27].

3 Literature Review

3.1 Challenges in Detecting Null pointer Dereferences:

Many tools have been created to deal with and detect NPDs. These tools, while effective in some cases, have issues.

The traditional approaches for detecting NPDs are split into two categories, static analysis and dynamic analysis.
Static analysis analyses the source code without executing it. Some static analysis techniques are abstract interpretation, data flow analysis and symbolic execution. NPDs are found by tracking the values of pointers and the flow of data in a program[28].

Dynamic analysis is when the program is executed and how it behaves is monitored. This includes runtime checks, fault injection and fuzz testing. This can help identify null pointer dereferences by detecting program crashes and unexpected behaviour[29].

False positives are a common issue with tools used to detect NPDs. This is when tools flag NPDs that do not exist, leading to a large number of warnings which take time for developers to address. One study found that the average precision of a selection of bug detectors was only 57.4%[30].

Pointer Aliasing occurs when multiple pointers point to the same memory location it can be difficult to assess whether a pointer is null or not. This can increase the number of false negatives in a system and reduce the overall accuracy of NPD detection tools. In this study [31] , pointer aliasing is credited with contributing to a certain NPD detection tool's false negativity rate of 89.81%. As this project is only using textual data it is not accessing memory locations of pointers. It is hoped that the issues of pointer aliasing will be mitigated by only focusing on textual data.

Path explosion is another issue that hinders the performance of NPD detection tools. Both static and dynamic analysis can be hindered by path explosion. This occurs when the computing power increases exponentially to verify complex software. Loops and recursion, exceptions, pointer aliasing and concurrency all contribute to the problem of path explosion. In this paper [32] , this issue was attempted to be remedied by developing a symbolic execution engine which limits the effects of path explosion. This project aims to detect NPD using textual information only. The models produced in this project will not suffer from path explosion issues.

Using ML models to analyse text data is also less computationally expensive than using traditional verification methods. As ML models do not have to compile programs before analysing the text data. Static analysis uses complex algorithms which can be very computationally expensive [33].ML models are also very scalable [34] and can be used in transfer learning, which cannot be said about traditional verification techniques. This study [35]proves that ML models can be integrated with static analysis tools.

3.2 Related Work

One study [36]investigates fixing Null Pointer Exceptions (NPE) in JAVA. This is done without using a test case. A model was created using random forest to predict the probability of an NPE. Statistical modelling is used to validate the patch. The random forest model is used to ensure that an NPE is less likely to occur after the patch is made. The path this study takes to solve the task at hand is fundamentally different to this project. The precision of the product produced in this study is 62%. The highest precision achieved by a model in this project was 82%, produced by XGBoost. It is important to note what these projects were trying to achieve and how they were evaluated is very distinct. Therefore the comparison of performance metrics is not an accurate measure of performance.

A study [37] investigating the use of deep learning methods to fix potential c errors. However, NPDs or NPEs were not investigated but this study rather attempts to fix simple coding errors like forgetting a closing brace in a function etc. These types of errors were not investigated in this project.

Research investigating [38] a novel patch generation system that aims to learn a probabilistic and application-independent model that corrects C code. This is done with a combination of defect localisation and patch generation. This study investigates machine learning methods for patch generation but uses other methods for defect localisation. This project is concerned with machine learning in defect localisation. This study does not focus on NPDs specifically but on a range of defects. One similarity between the study and the project at hand is the collection of open-source repositories with potential software defects.

One of the approaches taken in this project was to use a transformer and build upon Microsofts's CodeBert neural network. As discussed this neural net has been trained on a large corpus of code. In a paper [39] examining automatically generating the fix for bugs in JAVA using this neural network. The dataset used was called ManySStuBs4J. This study does not mention NPEs specifically. However, it does prove that certain bugs can be fixed in JAVA using the CodeBert network. Bugs were fixed in 17% -72% of the cases, depending on the type of dataset. This study investigates using a general-purpose CodeBert model that has been trained on 6 languages, Python, Java, JavaScript, PHP, Ruby and Go. Evidently, C is not one of these languages. There is a c-specific CodeBert neural net that will be used in the transformer investigation in this project.

In this paper [30], a transformer-based learning approach was used to identify false positive warnings. Infer, the static analyser developed by Meta was investigated in this study. The models developed in this project increased precision by 17.5% for Infers detection of NPDs. This project differs from the study at hand as it is not the intention of this project to reduce the false positivity of existing verification tools, but instead, to create an ML model that can be used instead or in combination.

Like this project, this study [40] aims to detect defects in C code using ML methods. The models produced in this project were trained using Oracle's internal dataset, Begbunch. This dataset is used to benchmark Parfait, a static analysis tool developed by Oracle. Like the project at hand, this study investigates detecting defects in C code functions. What separates these two projects is the data that was used to train models. The data used in this study is artificial as each code snippet is a few lines that illustrates the presence of a defect. The data in this project is real-world data that has been collected from open-source repositories. This study investigates a range of software defects. It has investigated NPDs but they are not the main focus of this study. The false positivity rate for NPD detection by the tool created in this study was 0% when tested on real-world data. The most performant model in this project produced a false positivity rate of 22% in NPD detection. It is believed that NPD detection was investigated less extensively in this study as 9 software defects were investigated, not just NPDs. There is also no mention of training transformers models in this study.

3.3 Limitations of Using Machine Learning Methods

The major limitation of this project is the lack of labelled data. This project is completely dependent on code being suitably commented on by developers. The efficiency, accuracy and complexity of machine learning tasks are significantly influenced by the quality of data used in training [41]. Bad commits being used in training and testing are unavoidable. A large amount of bad commit messages would cause the machine learning models to learn irrelevant features for the detection of null pointer dereferences and lead to the increase of false positives in testing. It is hoped that the vast majority of commits were done so properly. The most popular repositories on GitHub were used as it is hoped that the more popular repositories have proper commit conventions. It was proposed to generate labelled data using a static analysis tool like infer. The reason for not undertaking the task was primarily due to the considerable time investment and the inherent difficulty involved. As discussed static analysis tools also do not provide a complete remedy for null pointer exceptions.

Synthetic data will be produced in this project. This was created by artificially inserting NPDs into automatically generated C code files. This study [42]found that ML models produced using synthetic data are not less performant than models that use real-world data. However, due to time constraints and the scope of this project, synthetic data can never replicate the randomness and variability of the real-world data collected.
It is unavoidable for a machine learning model to flag some data points as positive when in fact they are negative. The study [39] that investigated detecting software defects in C code produced a model that performed well initially. However once investigated, it was found that

the model had poor precision rendering the models less performant than originally thought. If a model is produced with a high false positivity rate it will cause developers to spend a large amount of time investigating these false positives. This could deter users from utilizing the model.
One way of reducing the number of false positives in a model could be to use the fact that ML algorithms predict the probability that a data point belongs to a class. The threshold probability can be increased to prevent false positives, meaning the model must be more certain that an NPD is occurring to classify it as positive.

Machine learning models are uninterpretable. They are often called ”Black Boxes”, this is because it is very difficult to understand why they make a prediction. If a perfect model is produced, that correctly identifies an NPD 100% of the time. It may not be very useful as it would not be able to tell a developer why a null pointer a dereference is occurring, just that one is occurring. This study [43] states that there may be an overreliance on uninterpretable ML classifiers by researchers and students. An additional set of work may have to be applied after the model is trained and tested to decipher how it operates. It is unlikely that it will be clear how the models produced in this project operate without extensive research that goes beyond the scope of this project.

Based on the research it was decided to investigate SVM, XGBoost and neural networks in this project as there is a president for performant results in related tasks. Transformers will also be investigated, specifically CodeBert as it has proven to perform well for similar tasks throughout the literature.

4 Workflow and Plan

This project takes 2 routes, a traditional machine-learning approach and a transformer approach.
The traditional ML approach only uses real-world data to train and test the machine learning models. All models used were created for this project specifically and all text data is vectorised using the TF-IDF algorithm.
The transformer approach investigates using synthetic data and real-world data to train the model and exclusively real-world data to test it. This approach involves re-using an existing model that has been trained on specifically C code.

Data Collection:
All machine learning models need a large amount of data to learn the complex features of the task at hand. GitHub will be used to collect real-world data used to train and test the models produced in this project.

Create Test Set:
A test set will be created, comprised of different repositories than those used in training.

4.1 Transformer Approach

Genrate Synthetic Data:
Synthetic data will be generated using CSMith. CSmith is a tool that generates random C programs. This is was created to test compilers. Random C files will be produced. NPDs will then be artificially inserted into the data. This will create synthetic data that can be used to train the transformer models.
Use Bert's Auto-Tokeniser:
This is to represent the text data in a way that machine learning can understand. It preprocesses text data into an array of numbers as inputs to a model. This is used instead of TF-IDF. *neulab/codebert-c*'s word tokenizer will be used, this is a word tokenizer that has been trained on a large amount of text data.
Re-train Bert's Model on Synthetic Dataset:
The last layer of *neulab/codebert-c* is used for classification. The weights preceding layer will be frozen and the final layer will be trained on synthetic data to classify NPDs.
Re-train Bert's Model on Collected Dataset:

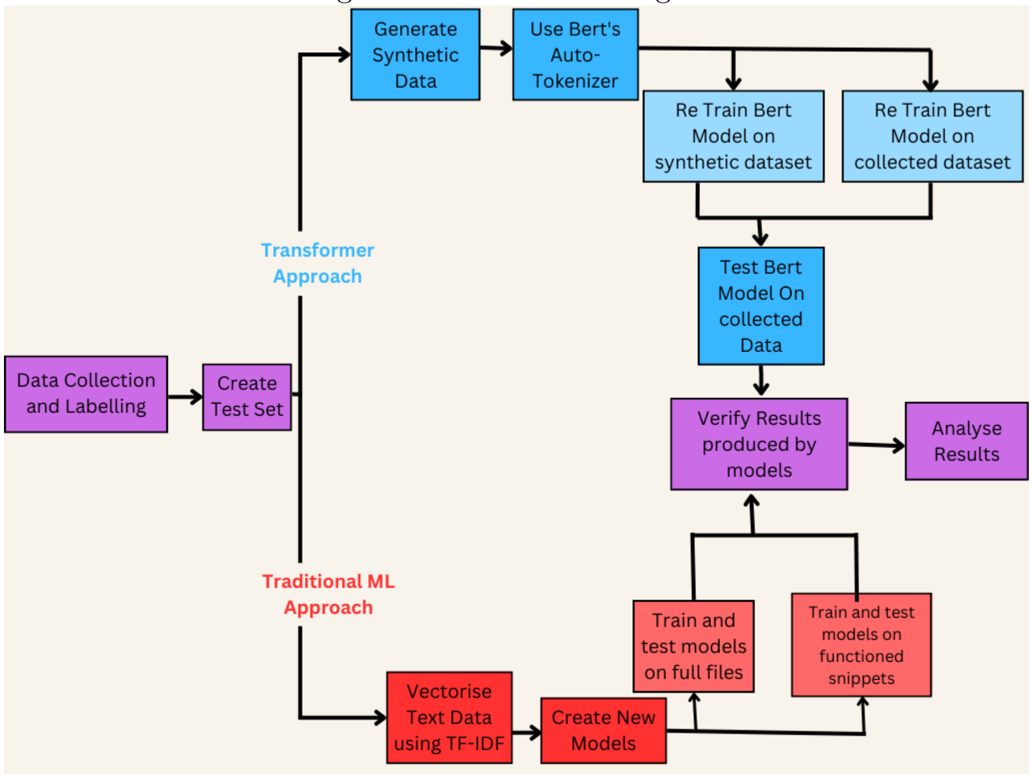
Again only the last layer of *neulab/codebert-c* will be trained. The real-world training dataset will be used to train the model.
Test Transformer Model on collected Data:
The transformer model produced will be tested on the real-world data collected.

4.2 Traditional Machine Learning Approach:

Vectorize the Text Data: Both the test and training set will be vectorized using the TF-IDF algorithm so the machine learning models can understand text data.
Develop Machine Learning Models: Many different machine learning algorithms will be investigated, XGBoost, Neural Networks etc. All algorithms have parameters that can be tuned to improve the models that they produce performance.
Train and Test Models on Full Files: The training data set will be used to train the ML models, and the test set will be used to test the ML models. Full C files will be used in training and testing.
Train and Test Models on functioned snippets: Only the functions that were altered in the commit will be used in the training and testing of the ML models

'Verify' Results produced by models: The most performant model from both the transfer learning approach and the traditional ML approach will be used in this section. Infer will be used to verify NPD in a selection of examples. True positive, True Negative, False Positive, and False Negative examples will be produced by the model.
Analysing Models: The models will be investigated and their strengths and weaknesses will be found. This will highlight the areas of improvement and help clarify what future work needs to be done to improve model's performance.

Figure 4.1: Workflow Diagram


Data Collection and Labelling
Create Test Set
Transformer Approach
Generate Synthetic Data
Use Bert's Auto-Tokenizer
Re Train Bert Model on synthetic dataset
Re Train Bert Model on collected dataset
Test Bert Model On collected Data
Verify Results produced by models
Analyse Results
Traditional ML Approach
Vectorise Text Data using TF-IDF
Create New Models
Train and test models on full files
Train and test models on functioned snippets

5 Data Collection:

A substantial amount of good-quality data is necessary data to train machine learning methods. The more complex a task is, the more data is necessary. Any natural-language processing problem needs a large quantity of data to teach the model the complex semantic features of the language. Although this project is not investigating human language this still applies. It was necessary to collect a large amount of C code snippets that contained NPDs and a large amount of C code snippets that did not contain NPDs. This was done using the GitHub API to collect open-source c code snippets. Commit points based on the fix of an NPD were collected.

5.1 The API query

The URL for the API query:

```
https://api.github.com/search/commits?q=null+pointer+dereference
&page={i}&per_page=100&language=c
```

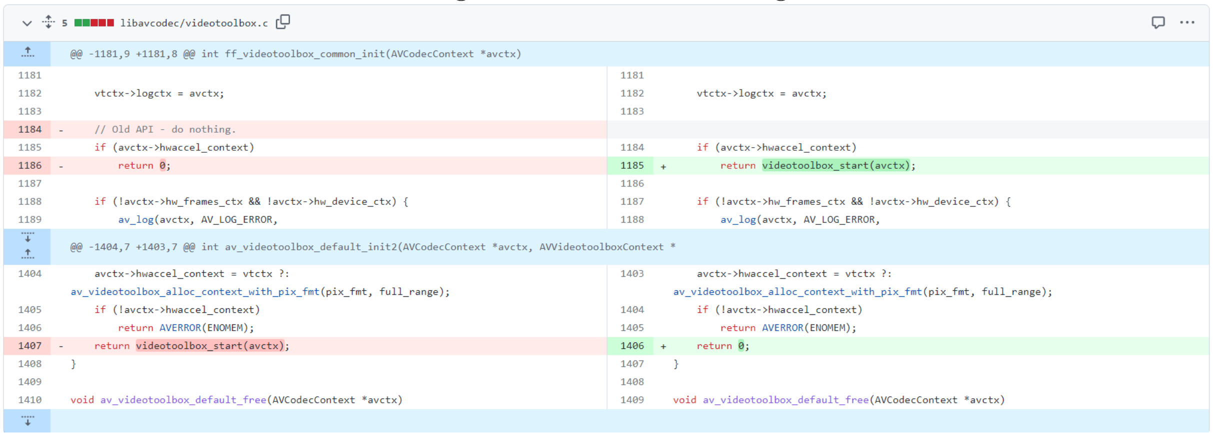
commits?q=null+pointer+derefernce queries commit messages with 'null pointer dereference'. Some examples would be "null pointer dereference fixed" or "null pointer dereference avoided". **&language=c** restricts the query to only include code that has been written in the language C. There are 100 commit points on any given page, **i**. In total, there are well over 30 million commit points available for collection on the API. However, there are many duplicates, as the same commit point appears in the query multiple times.
Due to a malfunctioning piece of code that tested to see if the commit point had already been queried, it was discovered in the later stages of the project that the database of commit points had a large number of duplicates. This meant that the database had to be recreated. The code that checked whether the API URL had already been queried was fixed. After this was done there were no longer any duplicate API URLs in the database. This significantly reduces the number of commit points available for collection and increases the time it takes to collect these commit points. More than 2 million commit points on the GitHub API were queried, there were many repetitions of commit points and just 2500 unique code snippets were collected. 385 different repositories were used to do this.

5.2 Data Labelling

As supervised learning is being investigated it is important to label all data that will be used in the training of the model. The theory and assumption that this project hinges on is a null pointer dereference occurs in the code snippet before the fix and does not occur in the code snippet after the fix. This image illustrates how the data was labelled:

Positive **Negative**

Figure 5.1: Data Labelling



```
5 libavcodec/videotoolbox.c
@@ -1181,9 +1181,8 @@ int ff_videotoolbox_common_init(AVCodecContext *avctx)
```

```
    vtctx->logctx = avctx;

-    // Old API - do nothing.
    if (avctx->hwaccel_context)
-        return 0;

    if (!avctx->hw_frames_ctx && !avctx->hw_device_ctx) {
        av_log(avctx, AV_LOG_ERROR,
```

```
    vtctx->logctx = avctx;

    if (avctx->hwaccel_context)
+        return videotoolbox_start(avctx);

    if (!avctx->hw_frames_ctx && !avctx->hw_device_ctx) {
        av_log(avctx, AV_LOG_ERROR,
```

```
@@ -1404,7 +1403,7 @@ int av_videotoolbox_default_init2(AVCodecContext *avctx, AVVideotoolboxContext *
```

```
    avctx->hwaccel_context = vtctx ?:
av_videotoolbox_alloc_context_with_pix_fmt(pix_fmt, full_range);
    if (!avctx->hwaccel_context)
        return AVERROR(ENOMEM);
-    return videotoolbox_start(avctx);
}

void av_videotoolbox_default_free(AVCodecContext *avctx)
```

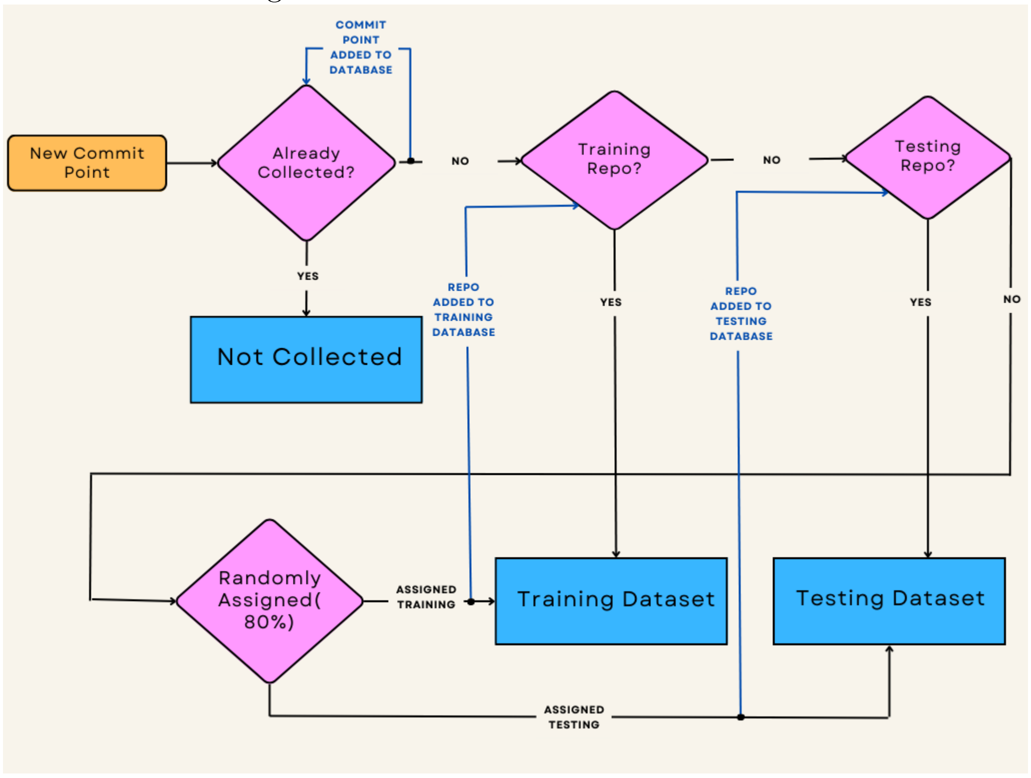
```
    avctx->hwaccel_context = vtctx ?:
av_videotoolbox_alloc_context_with_pix_fmt(pix_fmt, full_range);
    if (!avctx->hwaccel_context)
        return AVERROR(ENOMEM);
+    return 0;
}

void av_videotoolbox_default_free(AVCodecContext *avctx)
```

All real-world data collected in this project was labelled in this way.

5.3 Creation of Training and Testing Data Sets

A repository is a storage location of software. For this section, it is helpful to think of a repository as a collection of code snippets. A test set was created. This dataset was created from repositories which are different from those used for training to avoid bias in experiments due to project-specific programming conventions. This was around 20% of the size of the training set. 310 repositories were used to collect training code snippets and 75 repositories were used to collect testing code snippets.

Figure 5.2: How Data Sets were Created



1. When a new commit point is queried it is first checked whether it has been already been collected, if it has, it is not collected and the next commit point is queried. If it has not been collected the commit point is added to the database of all commit points so that it is not collected multiple times.

2. If it has not already been collected it is then checked whether the repository the commit point belongs to is a training repository. If the commit point belongs to a training repository, it is added to the training database.

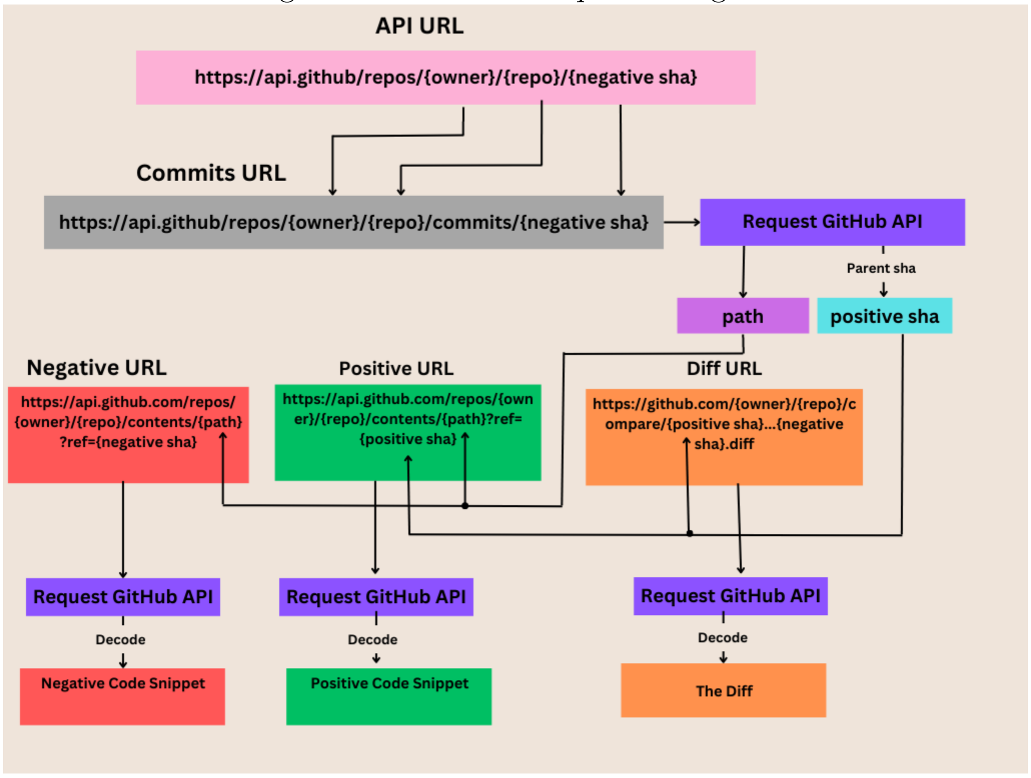
3. If the commit point does not belong to a training repository it is then checked whether it belongs to a testing repository. If the commit point belongs to a testing repository it is added to the testing database.

4. If the commit point does not belong to a training or testing repository then the repository that the commit point belongs to must be new to the project. This commit point is randomly assigned training or testing at a probability of 80% of training. The commit point then enters the training or testing database. The repository that this commit point belongs to is then added to either the training repository database or the testing repository database. Now, all commit points that belong to the same repository will be added to the same dataset(training or testing) as S said repository.

5.4 Finding Code Snippets

The GitHub API was once again used to find code snippets that corresponded to the commit points. The 'sha' in GitHub is a value that identifies the specific changes to a repository and when they were made. The github api url follows the form **https://api.github/repos/{owner}/{rep**
. The sha values correspond to the code **after** the commit. Therefore in this specific case, the GitHub API is returning the sha value after the NPD was fixed by a developer. Therefore the sha corresponds to the code that does not contain an NPD. This was named the negative sha. A parent sha is where the current sha branched off from. The parent sha of the negative sha is therefore the sha that corresponds to the code that contains an NPD, the 'positive' sha.

Figure 5.3: GitHUb Requests Diagram


API URL
https://api.github/repos/{owner}/{repo}/{negative sha}
Commits URL
https://api.github/repos/{owner}/{repo}/commits/{negative sha}
Request GitHub API
Parent sha
path
positive sha
Negative URL
https://api.github.com/repos/{owner}/{repo}/contents/{path}?ref={negative sha}
Positive URL
https://api.github.com/repos/{owner}/{repo}/contents/{path}?ref={positive sha}
Diff URL
https://github.com/{owner}/{repo}/compare/{positive sha}...{negative sha}.diff
Request GitHub API
Decode
Negative Code Snippet
Request GitHub API
Decode
Positive Code Snippet
Request GitHub API
Decode
The Diff

The GitHub API returns the value of the most recent commit 'sha'

- The GitHub API returns the 'API' URL for the commit point. This was used to find the commits URL.
- The commits URL is then used to request the GitHub API.
- The response from the GitHub API is used to find the path to the file that was altered. The parent or 'positive' sha was also found using the response from the commit URL.

- The contents URL was found for both the negative and positive URLs. This can be seen in the diagram. The diff url was also found using the negative and positive sha. The repo and owner values were constant for all URLs.
- The contents urls and the diff URL were used to request the GitHub API.
- The responses from the GitHub API were then decoded using the function **b64decode** from the **base64** library in Python.
- The decoded content was then saved as a text file on a local drive on a computer.

Entire C files were saved from the contents URL.

5.5 'Functioning' Code Snippets:

The code snippets returned from the contents URL were entire C files. It was hypothesised that having such large snippets would not highlight the features that cause NPDs. Therefore another database was created which contained only the functions that were altered based on the commit point to fix the NPD. For both the training and test set only the C functions that were altered at each commit point were kept, with surrounding code being dropped.
The diff was used to find the functions that were altered to fix NPDs.
All C functions had to be properly formatted with each open bracket '{' needing a corresponding closing bracket '}'. There were many examples of the commit point corresponding to a change outside a function definition. If the code was not formatted like this it would be dropped and not used in training or testing.
Typically these commit points altered a single function but a small number of commits involved changes to more than one function.

5.6 Generating Synthetic Data

To investigate transformer models a lot of data is necessary due to the large number of parameters that need training. It was unfeasible to collect this large quantity of data using the GitHub API and real-world code. Therefore it was proposed to create synthetic data.
Csmith is a test case generation tool that generates C programs. Once programs were generated, functions would be found. These functions were assumed to be NPD-free. The functions would be saved in their original form as negative examples. A very simple NPD would be inserted into the function, and then this would be saved as a positive example.
Example of Synthetic Data Generation:

```
static int32_t * func_116(int32_t * const  p_117, uint8_t  p_118)
{ /* block id: 29 */
    uint64_t l_136 = 0UL;
    int32_t *l_179 = &g_91;
    int32_t *l_179 = &g_91 ;
    int32_t *l_180[1];
    return l_180[0];
}

static int32_t * func_116(int32_t * const  p_117, uint8_t  p_118)
{ /* block id: 29 */
     uint64_t l_136 = 0UL;
      int32_t *l_179 = NULL;
     int32_t *l_180[1];
      *l_179 = 582820;

     return l_180[0];
}
```

These code snippets are based on real code snippets in the synthetic database, that have been altered to highlight the synthetic data generation process.
The code above demonstrates how all synthetic data was produced. The memory address of a random integer pointer variable is changed to equal NULL. This variable then goes on to dereference to a random integer in the remaining lines of code. While this does produce synthetic data it is not high-quality and does not have enough variability. There are many ways an NPD can occur. A better way of generating synthetic data would be to alter more complicated real-life code and use verification techniques to check for NPDs and assign labels, or to use real-world code found on GitHub and artificially create NPDs. This goes beyond this project's scope and synthetic data was created as a proof of concept rather than to make high-quality data.

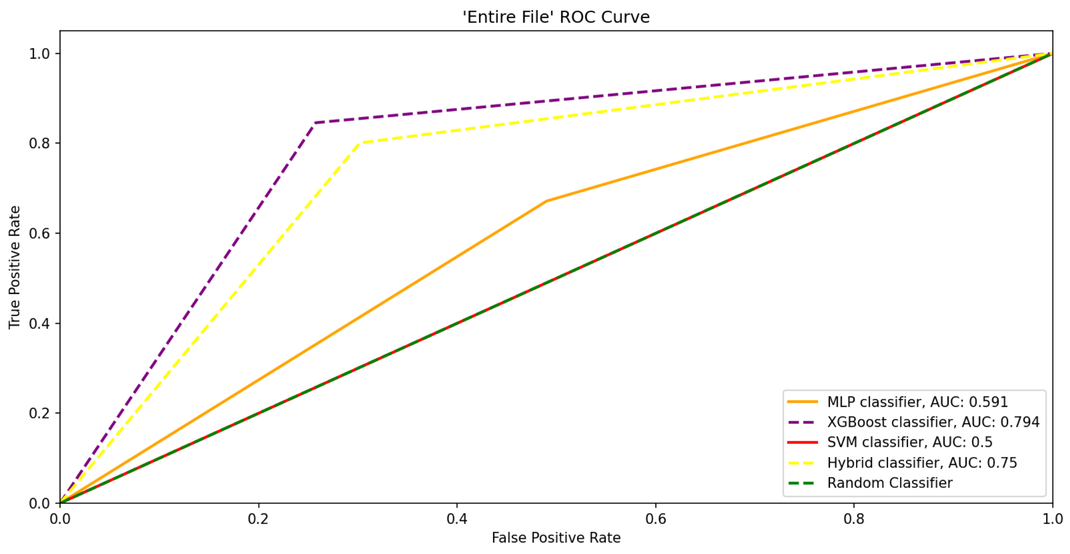
6 Traditional Machine Learning Approach, Results and Discussion

This section aims to display the results of the traditional machine-learning approach and comment on their performance, as well as evaluate their limitations.
The phrase data point will be used. In this context, this means a code snippet that has been labelled positive or negative.
All of the text data in this section was vectorized using the TF-IDF algorithm. Only real-world data collected from GitHub was used in training and testing in this section.

6.1 Entire File Data Results

This section investigates using entire C files to train the machine learning model. There were around 1800 code snippets used in training and 400 code snippets used in testing. This vectoriser was generated using training data and the *fit* function and was applied to both the training data and testing data using the *transform* function.
All models in this section used the same training and testing datasets. The results from this section are less performant than the results from functioned data training. Therefore entire file models will be investigated to a lesser degree than functioned data training.

6.1.1 ROC curve



6.1.2 XGboost

The XGBoost model has surprisingly performant results with an AUC of 79% The maximum depth of the decision trees was 6, the learning rate was 0.3 and the number of boosting iterations was 100. The exact hyperparameters can be seen in the appendix below. This was the most performant model in this section and produced impressive results given the circumstances. XGBoost produced a very impressive model considering such large amounts of irrelevant data were present in training and testing

Accuracy / AUC	79%
Recall	84%
Precision	76%
F1 Score	78.7%

6.1.3 MultiLayer Perceptron

MultiLayer Perceptron did not perform as well as the XGBoost model. The neural network had one hidden layer with 100 units with the activation function ReLU. These results are not very performant considering a random classifier would produce an AUC of 50%. The large amount of irrelevant data causes the model's weak performance.

Accuracy / AUC	59%
Recall	67%
Precision	57%
F1 Score	61.5%

6.1.4 Support Vector Machine

The SVM model was linear with a kernel size of one. SVM did not produce performant results. In fact, the results produced by SVM were worse than a random binary classifier. The high value of recall is due to the model automatically assigning every data point as positive. Situations like this highlight the importance of having multiple metrics when evaluating a model's performance.

Accuracy / AUC	49%
Recall	98%
Precision	49%
F1 Score	65.3%

6.1.5 Hybrid Learning

The Hybrid model uses two machine learning models, MLP is used to find the likelihood that a data point belongs to a class and then this is immediately followed by XGBoost which classifies which class a point belongs to. There are 3 hidden layers of 150, 75 and 50 units, all with ReLu activation. The Xgboost model has a maximum depth of 5, a learning rate of 0.3 and the number of boosting iterations was 100. The results from the hybrid learning model were less performant than the XGBoost model, yet more performant than the MLP and SVM. The results from this model, like the XGBoost model, are surprisingly performant due to the large amount of irrelevant data. The metric results produced in this model are consistent with each other with a range of just 8%.

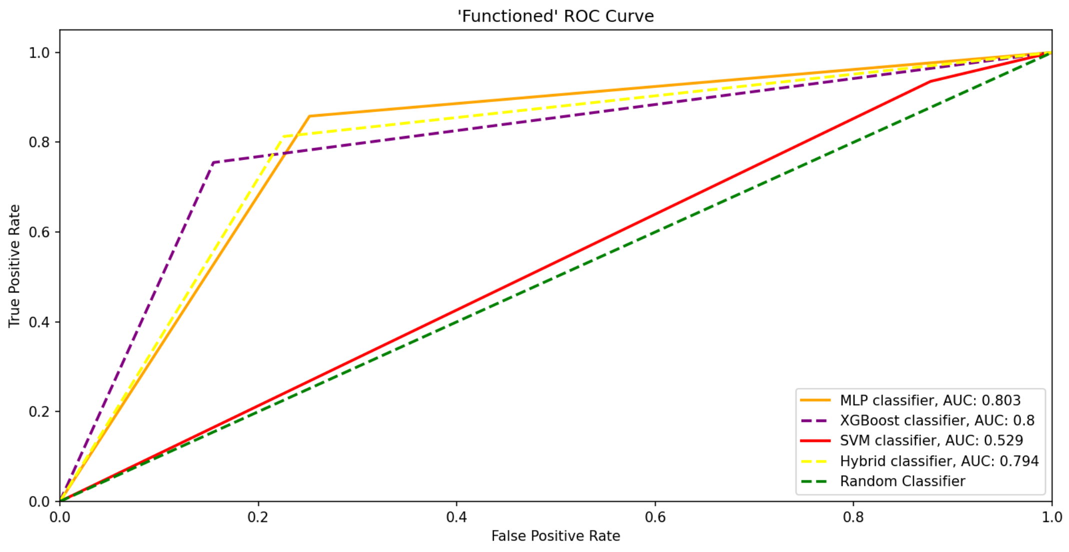
Accuracy / AUC	75%
Recall	80%
Precision	72%
F1 Score	75.78%

6.2 Functioned Data Results

There are just over 1200 data points in the training set and 300 data points in the testing set.

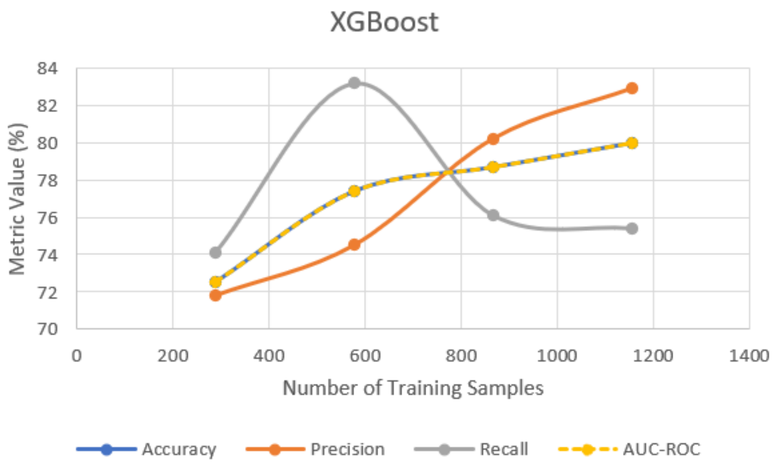
6.2.1 ROC-curve

The ROC curve for all metrics was produced. It can be seen that SVM was the least performant metric with a performance that is marginally better than a random classifier. Therefore there will not be an in-depth discussion on how it performed in relation to training data. Like entire file training, SVM had a kernel size of 1. This was to reduce the computational complexity to speed up training time. XGBoost, MLP and the hybrid classifier all have very similar performances with an AUC of around 80%.



6.2.2 XGBoost

The graph of metric performance vs the number of training samples can be seen below. This XGBoost model had identical hyper-parameters as the entire file machine learning model. The XGBoost classifier performs as expected, where performance metrics increase as the number of training samples is increased. The AUC and accuracy are very closely linked, therefore share an almost identical curve. Xgboost trains very quickly compared to other algorithms and produces an accuracy of 80% and a precision of 82.9% when trained on 1200 snippets. If more data were available to increase training and testing test XGBoost could produce a simple yet effective model. As can be seen, the recall values raises to a high level and then drops down, this is most likely due to the model assigning most data points as positive. Once the model learns to assign some data points as negative the recall value drops, yet the precision and AUC value increase.

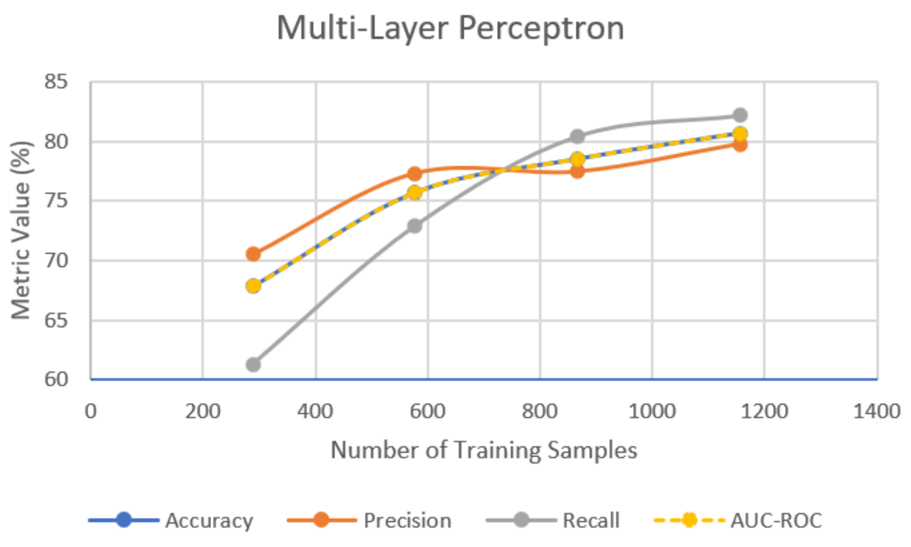


Results at max training samples:

Accuracy / AUC	80%
Recall	75.6%
Precision	82%
F1 Score	78.67%

6.2.3 Multilayer Perceptron

Multilayer perceptron is a type of neural net. There were 100 units in a single hidden layer of the neural net with the activation function ReLu. Like the XGBoost model the training metrics increase as the number of training samples increases. The AUC and the accuracy curve have a stronger upward trajectory than the XGBoost model. This implies that if more data was added to the training set the metrics would improve. This also had relatively performant results with an accuracy/AUC of 80.64% and precision of 79.7%. While MLP may appear slightly less performant than Xgboost it is important to note that the value for recall at 1200 is 82.5% for the MLP and just 75.4% for XGBoost. The metric values in MLP are more consistent with each other and tend to increase in a similar fashion as training samples increase, which cannot be said about the Xgboost model.

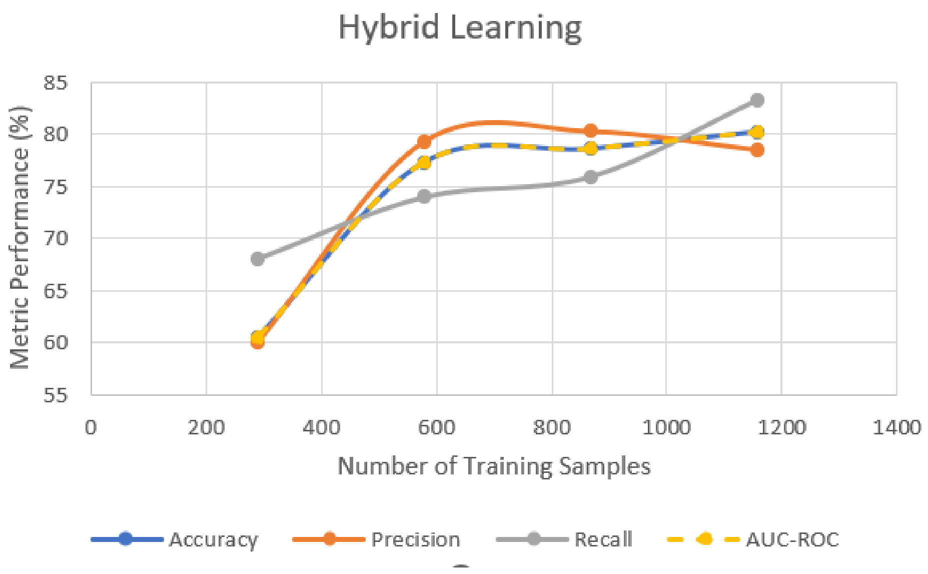


Results at max training samples:

Accuracy / AUC	80.6%
Recall	82.2%
Precision	79.7%
F1 Score	80.9%

6.2.4 Hybrid Learning Approach

The 'functioned' data approach uses the exact same configuration and hyperparameters as the entire file training. As can be seen in the AUC/accuracy curve the hybrid model learns quickly and plateaus quickly as well. Therefore it is hypothesised that if the model was trained on more data, the metrics' values would not increase significantly. The value of AUC/accuracy is 80.21%, the value of precision is 78.5% and the value of recall is 83.22%.



Results at max training samples:

Accuracy / AUC	80.2%
Recall	83.2%
Precision	78.5%
F1 Score	80.8%

6.3 Discussion of Traditional ML Approach Results

It is believed that the most performant model of the traditional ML approach is 'functioned' data MLP as it had the highest values for AUC and F1 with 80.6% and 80.9% respectively. By some measures, the XGBoost model would be the most performant model produced as it has the highest precision score. False positives should be avoided. However, it is of the opinion that the higher AUC and F1 score of the MLP model trumps the high precision value of the XGBoost model. As MLP is a deep learning approach it also lends itself to being used in transfer learning. Therefore this model will be investigated in the verification and analysis section of this report.
The results of traditional ML approaches were very performant considering the models only take the occurrence of words into account when making their prediction, not word order. Although the results are quite performant given the amount of data available. It is important to remember that word order plays a huge role in how code operates. This can be seen in the toy examples in the TF-IDF paragraph in the relevant materials section of this report. The models must be analysed to a sufficient degree to ensure they are operating as expected. It is also very difficult to say for certain that they are in fact detecting NPDs, it is possible that there is a pattern in the data that humans may not be able to identify, yet possible for a machine learning model to pick up on. For example, it is more likely for there to be more

code words in a function after a commit is made, therefore the model could be more likely to classify a data point with more text data as negative if it has more words.

The models produced in this section cannot take word order into account so, therefore, cannot create a complex model and will never truly be able to understand why an NPD is occurring. The models produced in this section are not useless as they prove that it is possible for machine learning models to detect NPDs to a certain degree. They also provide simple and effective models. However, they do not produce models that can be used in transfer learning, to produce a model that is extremely accurate at detection or to produce a model that can fix NPDs. While the models yield impressive results, how they have been trained is inherently flawed if they are to be used to produce an operational product.

7 Transformer Approach, Results and Discussion

The results produced in the transfer learning approach were less performant than the results produced in the traditional ML approach. 2 models were produced using this method. All parameters in the *neulab/codebert-c* were frozen and only the last layer, which is used for classification, was trainable. This is the typical approach when re-using existing neural networks for classification. There are 124,647,170 parameters in the model. There are 1,538 parameters in the classification layer. This means that 124,645,632 were frozen. Freezing all layers except the classification layer hopes to greatly reduce the quantity of data necessary for training as there are fewer parameters to train.
The transformer approach used 'functioned' data, not entire C files.

7.1 Results

7.1.1 Real World Data Model

The real-world data model produced results that were not performant, the classifier simply classified every data point as positive. This model learned no features that can cause NPDs.

Accuracy	50%
Precision	50%
Recall	100%
F1 Score	66.6%

7.1.2 Synthetic Data Model

The model was trained on the synthetic data that was produced. The model was tested on the data frame that was used to train the real-world model. It is important to test models that have been produced by synthetic data on real-world examples, if possible. This is to ensure the model performs as well 'in the wild' as it does in testing. Like the real-world data model's results, the results for the synthetic data training were not as performant as the traditional machine learning methods. The synthetic data model has similar, unperformant results to the real-world data model.

Accuracy	50%
Precision	50%
Recall	90.5%
F1 Score	64.4%

7.2 Discussion of Transformer Approach Results

As discussed transformers take word order into account. This is why it is vital to investigate a transformer approach in this project. However, neither of the transformer models managed to extract features that cause NPDs.
The main limitation of these models is the lack of available data. Transformer models are more complex and therefore need much more data than traditional ML models. There is not enough easily accessible real-world data on GitHub. As can be seen from the results of the real-world data training, the data that is easily accessible is not of sufficient quantity to build performant transformer-based ML models. Therefore synthetic data must be investigated.
The quality of synthetic data produced was subpar. If this route were to be investigated in the future, the variability of synthetic data would need to be increased to a sufficient standard. There are numerous ways of dereferencing a NULL pointer, only assigning a null pointer to an integer was investigated: **nullPointer=integerValue*. For example, a null pointer could be assigned the return value from a function: **nullPointer =function()*, or could be assigned to an array value: **nullPointer=array[index]*, these are just a few examples of dereferencing null pointer. A larger variety of NPDs would need to be generated to produce an effective transformer model.
The synthetic data produced in this project did not take fixes of NPDs into account. The real-world code collected focused entirely on NPDs which had been identified and corrected. The before and after versions of these NPDs were suitable for training. Additional work would be required to produce high-quality synthetic data.
The resources and data to build transformer-based ML models were not available. It is believed that transformer models could produce excellent tools for detecting NPDs, this however goes beyond the scope of this project.

8 ’Verification’ and Analysis of Model’s Performance

As discussed the model that will be verified and analysed in this section is the ’functioned’ data MLP.

8.1 Verification

Test data was created to verify the model’s performance. Infer, the static analysis tool developed by Meta was used to find the true state of each example. If infer reported an NPD it would be labelled positive and if infer did not report an NPD it would be labelled negative. The MLP model was used on the test examples to make predictions.
Infer must be able to compile the program before checking for NPDs. This is why the real-world test set was not used to verify results as there would be other functions, global variables and more which would not be included in the code snippets, causing it not to compile.

		Model Prediction	
		p'	**n'**
Infer Prediction	**p**	True Positive	False Negative
	n	False Positive	True Negative

8.1.1 True Positive

```
void example() {
    int* ptr = NULL;
    *ptr = 42;
}
```

In this example, both the ML model and Infer detect an NPD. While the results are as expected, the model can be tricked. Changing the integer *ptr was dereferenced from *42* to *78* does not affect whether an NPD is occurring. Yet, once this change was made the model assigned the code snippet negative, producing a false negative. This is a large limitation of machine learning, it is very difficult to determine why decisions are being made. This highlights the importance of extensive testing. To fix this issue more training data is needed to teach the ML model that when an integer pointer is dereferenced, the number it is dereferenced to does not affect whether an NPD is occurring. Interestingly, when using the macro definition of null, *((void*)0)*, the model's prediction remained positive. Therefore, in training, the model may have learned that the two are synonyms.

8.1.2 True Negative

```
void example(){
int *ptr1 = NULL;
int num = 10;
if (*ptr1 != NULL){
  *ptr1 = num;
}
}
```

In this example both the ML model and infer managed to realise that the NPD that would have occurred was avoided by an *if not null* statement. If this statement is removed the model assigns the code snippet as positive. This bodes well for the model's performance as it demonstrates that the model has learned that *if not null* statements are likely to avoid NPDs.

Altered code snippet:

```
void example(){
int *ptr1 = NULL;
int num = 10;

if (*ptr1 != NULL){
  num = 11;
}
*ptr1 = num;
}
```

In this example, the *if not null* statement does not affect the pointer dereference. The model assigned this code snippet negative, meaning it did not pick up on an NPD. Infer detected an NPD. This indicates that the model is simply checking for the existence of if,! and NULL in the code snippets. This highlights the importance of creating a functional model that takes word order into account.

8.1.3 False Positive

```
void example(){

int* ptr1 = NULL;
int* ptr2 = ptr1;
int value = *ptr2;
}
```

The model assigned this model as positive, yet Infer did not pick up on an NPD. The variables *ptr1* and *ptr2* are both assigned NULL yet neither is dereferenced. To avoid false positives like this word order would need to be taken into account. The words in this code snippet could potentially be re-ordered to create a code snippet that would cause an NPD.

8.1.4 False Negative

```
void example () {
int * ptr = NULL;
* ptr = 3 ;
return ;
}
```

The model classified this snippet as negative. Infer detected an NPD. Like the false positive this issue could potentially be solved with a model that takes word order into account. I

8.2 Analysis

While the model performs well and assigns the correct label the majority of the time, it is important to evaluate where it does not. Due to the time limitations of this project, the test examples were not too complex. Machine learning models are often regarded as 'black boxes' which means it can be difficult to understand their prediction. As in the earlier example **8.1.1** a small change, which didn't impact the NPD resulted in an alternative prediction. Creating more complex examples to test the model, would be an important step in future work. An operational synthetic data generation method could help in producing more complex examples which could be verified using a static analyser.
The model is far more likely to assign code as negative if an *if not null* statement exists in the code snippet this can be seen from example **8.1.2**. It is possible, that the data collection and labelling process has produced a model that inversely detects NPD fixes, ie the model assigns negative if it believes that an NPD has been fixed. This is due to all negatively labelled code snippets containing an NPD which were fixed. There were no instances of code that lacked an NPD used in training or testing. If this is the case this model would be useful as it could find NPD fixes in open-source repositories. This may make it possible to create a database that would be large enough to train transformer models which could be used to teach an ML model to fix NPDs. The model could also be used in transfer learning when training this collected data.
Using infer to verify a model's performance has issues. As discussed, infer, like all static analysers, does not have an accuracy of 100%, it does not identify every NPD. The findings obtained from static analysis tools should not be regarded as infallible, however, it was assumed that infer would be far more accurate than an ML model still in its infancy. Therefore a combination of dynamic analysis, static analysis, developers' comments on GitHub

and code analysis should be used to verify whether an NPD is occuring once the model is more advanced.

9 Future Work and Ethics

9.1 Future Work

Performance metrics improve once irrelevant data is dropped from the training and testing datasets. This can be seen from the 'functioned' models having more performant metrics than the 'entire file' models. This is very pronounced in the models that use neural nets. This lends itself to using static program slicing to help shed irrelevant data. A static slice of a program contains all statements that may affect the values of a variable at any point in the program[44].This could be used to alter the database to only contain the code affected by the commit point, therefore only containing code that would contribute to the fix of a null pointer dereference. Frama-c is a static analyser for c code that could be used to do this. This would be applied to both the training and testing set and transform and condense the code snippets. The product produced by this method would apply frama-c to the line of code to be investigated, and produce a small condensed snippet to be analysed by the model. This could be used in both a traditional ML approach and a transformer approach. Reducing the code snippets to a smaller size may make word order play a smaller role in the detection of NPDs. This may produce a more dependable traditional ML model as word order would contribute to a lesser degree.

The machine learning models, aside from the synthetic data model, have been trained and tested on data that has been collected from one source. It is important to collect data from different sources to increase variability. There were 375 different repositories used for training and testing, this is relatively small. It could lead to repository-biased results. There are around 30 million commit points that relate to the message "Null Pointer Dereference" on GitHub. Many of these are duplicates. It is difficult to find the true amount of unique commit points that are available for collection.

Creating a large amount of good-quality synthetic data would be essential in creating a model that can train transformer models effectively. This could be done by altering collected data so it alters variable names, and statement positions, yet conserves the code's operation and NPDs. Another way of creating synthetic data would be to use open-source c repositories to collect code snippets and artificially insert NPDs.

Often static analyser developers have large collections of labelled test cases used to benchmark tools. Working with the static analyser developers may be an option to gather data.

Once a model performs well in testing it would be important to test the model with real-world code. This would be beneficial for two reasons. Firstly, the model could potentially detect NPDs that have been laying dormant in code. Secondly, it could also highlight a model's true performance by seeing how many false positives it produces. As the model was trained on an equal number of positive and negative examples, there is bias in the training set. A real-world code snippet is far more likely to not have an NPD than have an NPD. If the model constantly flags NPDs that do not exist the model may need to be retrained with a less biased training set.

Once the model has been tested on real-world code to a sufficient degree, the model could be integrated with a static analysis tool, like Infer. The tool produced would be able to quickly scan a C file using the model and flag any potential NPDs. Infer could then test for NPDs verifying the results produced by the model. The strengths of both tools could be leveraged to make an extremely accurate NPD detector, that is able to scan entire repositories very quickly.
To make the model more accessible to developers a user interface could be made. This could be done by integrating the model into an existing IDE or a web application. The UI would allow users to input code snippets and receive information and the likelihood of it containing a null pointer dereference error. Allowing a user interface like this would allow the model to collect more data. This data could be used to improve the model's performance.

The model could be applied to other programming languages. One relatively simple way of doing this would be to create a model that finds NPDs in C++ code as well as C. To collect real-world data for this, the GitHub API query would be altered from **&language=c** to **&language=c&language=cpp**. This would extend the API query to collect both C and C++.

As discussed in the related work section there are multiple examples of work in a similar area. The findings from both of these projects could be integrated to produce a very performant resulting model.

Build on the model produced and use a transfer learning approach to create a model that aims to fix null pointer dereference errors by using generative transformers. The original model produced could be used in both data collection and transfer learning to help create a model that fixes NPDs.

9.2 Ethics, Security and Privacy

It is not immediately obvious what the ethics, security and privacy concerns of this project are. The machine learning models were trained and tested on code from open-source repositories. This data is publicly available, therefore there are no privacy concerns in relation to training data. Once the project develops out of its infancy stages security and privacy issues will arise.

As discussed once this machine learning model is tested to a sufficient standard it is hoped to host it publicly on a user interface. This could involve integrating the model into an existing IDE or a web application. As machine learning models need lots of data to train and test their models, machine learning-based applications capture user data to improve the model's performance[45]. The UI that hosts this model would most likely use the same mechanism to collect user data to retrain the machine learning model in the hopes to improve it in future. This could cause privacy concerns as closed-source software can contain very sensitive information, especially if this information is leaked or stolen. Many applications like this one do not charge for their services but instead, sell the data that the users provide to a 3rd party. This article [46] discusses the ethics of third-party advertising and whether it is ethical for companies to take part in targeted ads.

The model in this project used Microsoft's CodeBert pre-trained neural net in transfer learning. From this research [47] it can be seen that code models are vulnerable to backdoor attacks. Consequently, the models built upon CodeBert can produce pre-defined malicious outputs on examples that trigger the back door. This paper presented that state-of-the-art defence methods cannot provide sufficient protection for code models. This is a very serious security threat that would need to be addressed if the model was to be hosted publicly. The model could produce security threats and vulnerabilities, the very thing it is trying to remedy.

A machine learning model that can detect NPDs to a high standard could be used to analyse code much faster than current verification tools as the code does not need to be compiled. If a bad actor had access to a victim's code base, whether it's open-source or by other means, the bad actor could use the ML tool to quickly scan for NPDs. A bad actor could leverage the vulnerabilities that NPDs have the potential to cause. To mitigate against the use of the tools produced in this project the developers would need to implement proper access controls. This means that only authorized users are allowed to use the tool. How users use said tool would also need to be logged, in to track and ensure that the tool is not being misused. Ethical guidelines for the tool would need to be created and users must agree to them before using the tool.

10 Conclusion

In conclusion, this project laid the groundwork for using ML methods for detecting NPDs. While a fully functional model was not produced, this project took foundational steps for development in this area.
This project highlighted the difficulties associated with NPD detection, as well as how they could be potentially fixed by ML. The limitations of ML have been explored, evaluated and analysed throughout this project.

A variety of ML algorithms have been explored throughout the project. The performance of each model has been quantified and their strengths and limitations examined. Evidently, it is possible for ML models to learn key features which contribute to the existence of NPDs. Given the available data, relatively performant models have been produced.
The major limitation of this project was the lack of labelled data and the time-consuming approach of collecting data from GitHub. The progress of this project was impeded by issues in data collection that resulted in thousands of code snippets being collected multiple times. These duplicate snippets were used in both training and testing. This skewed the initial results produced by models. Once this issue was remedied all data collection had to be re-done and all models produced needed to be retrained and retested. It is suggested to only undertake a similar project in future if there is access to a large database of NPDs. Using GitHub to collect NPDs is not an efficient data collection strategy. Developers of verification tools have internal databases of software defects. Working with these developers and accessing these databases would be of great benefit to produce a functional model.
The information learned during this project suggests that the next stages should focus on generating synthetic data and collecting real-world data to train and test the transformer models. This would be an improvement in the work done with the traditional ML models.

This project serves as a stepping stone in the direction of detecting and potentially fixing NPDs using ML methods. This investigation has provided guidance, knowledge and instruction for future work in this area. This project also laid the groundwork for future projects to investigate alternative software defects, not specifically NPDs.

11 Bibliography

1. Hoare, Tony (2009). "Null References: The Billion Dollar Mistake" (Presentation abstract). QCon London. Archived from the original on 28 June 2009.

2. Jin, Wenhui, et al. "NPDHunter: Efficient Null Pointer Dereference Vulnerability Detection in Binary." IEEE Access, vol. 9, 2021, pp. 90153–90169

3. Introduction to Formal Verification, Berkeley University of California, Retrieved November 6, 2013

4. "RV'01 - First Workshop on Runtime Verification". Runtime Verification Conferences. 23 July 2001. Retrieved 25 February 2017.

5. "Runtime Verification for C Programs with Concurrency and Pointer Arithmetic" by Y. Cheon and J. T. Kim (2017).

6. Mladenić, Dunja, et al. "Feature Selection Using Linear Classifier Weights." Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 25 July 2004

7. Noble, William S. "What Is a Support Vector Machine?" Nature Biotechnology, vol. 24, no. 12, Dec. 2006, pp. 1565–1567.

8. Hofmann, Martin. Support Vector Machines -Kernels and the Kernel Trick an Elaboration for the Hauptseminar "Reading Club: Support Vector Machines." 2006.

9. Priyanka, N.A., and Dharmender Kumar. "Decision Tree Classifier: A Detailed Survey." International Journal of Information and Decision Sciences, vol. 12, no. 3, 2020, p. 246

10. Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785-794). ACM

11. Zhang, Y., Wang, C., Huang, W., & Liu, T. (2020). Human activity recognition with wearable sensors using XGBoost. IEEE Access, 8, 10577-10588.

12. Fathi, Ehsan, and Babak Maleki Shoja. "Chapter 9 - Deep Neural Networks for Natural Language Processing." ScienceDirect, Elsevier, 1 Jan. 2018

13. Bre, F., Gimenez, J.M. and Fachinotti, V.D. (2018) Prediction of Wind Pressure Coefficients on Building Surfaces Using Artificial Neural Networks. Energy and Buildings, 158, 1429-1441.

14. Günther, Frauke, and Stefan Fritsch. “Neuralnet: Training of Neural Networks.” The R Journal, vol. 2, no. 1, 2010, p. 30

15. Jurafsky, Dan, and James H. Martin. “Speech and Language Processing.” Stanford.edu, 2018

16. LIANG Jie, CHEN Jiahao, and CHEN Jiahao LIANG Jie. “One-Hot Encoding and Convolutional Neural Network Based Anomaly Detection.” Journal of Tsinghua University(Science and Technology), vol. 59, no. 7, 21 June 2019, pp. 523–529,

17. Ramos, Juan. Using TF-IDF to Determine Word Relevance in Document Queries.

18. Yang, Qiang, et al. Transfer Learning. Google Books, Cambridge University Press, 13 Feb. 2020

19. Medsker, Larry, and Lakhmi C. Jain. Recurrent Neural Networks: Design and Applications. Google Books, CRC Press, 20 Dec. 1999

20. Pascanu, Razvan, et al. “On the Difficulty of Training Recurrent Neural Networks.” Proceedings.mlr.press, PMLR, 26 May 2013

21. Chung, Junyoung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2014. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.” In NIPS 2014 Workshop on Deep Learning, December 2014.

22. Vaswani, Ashish, et al. “Attention Is All You Need.” ArXiv.org, 2017.

23. Tay, Yi, et al. “Synthesizer: Rethinking Self-Attention for Transformer Models.” Proceedings.mlr.press, PMLR, 1 July 2021

24. Feng, Zhangyin, et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages.” ArXiv:2002.08155 [Cs], 18 Sept. 2020

25. Li, Liuwu, et al. “A Hybrid Model Combining Convolutional Neural Network with XGBoost for Predicting Social Media Popularity.” Proceedings of the 25th ACM International Conference on Multimedia, 23 Oct. 2017

26. Bradley, Andrew P. ”The use of the area under the ROC curve in the evaluation of machine learning algorithms.” Pattern recognition 30.7 (1997): 1145-1159.

27. Narkhede, Sarang. ”Understanding auc-roc curve.” Towards Data Science 26.1 (2018): 220-227.

28. Kaur, Arvinder & Nayyar, Ruchikaa. (2020). A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. Procedia Computer Science. 171. 2023-2029. 10.1016/j.procs.2020.04.217.

29. Bell, Thoms. “The Concept of Dynamic Analysis.” ACM SIGSOFT Software Engineering Notes, vol. 24, no. 6, 1 Nov. 1999, pp. 216–234

30. Kharkar, Anant, et al. “Learning to Reduce False Positives in Analytic Bug Detectors.” Proceedings of the 44th International Conference on Software Engineering, 21 May 2022 Accessed 27 Mar. 2023.

31. Jin, Wenhui, et al. “NPDHunter: Efficient Null Pointer Dereference Vulnerability Detection in Binary.” IEEE Access, vol. 9, 2021, pp. 90153–90169, ieeexplore.ieee.org/stamp/stamp.j

32. Koppier, Stefan. The Path Explosion Problem in Symbolic Execution an Approach to the Effects of Concurrency and Aliasing. 2020.

33. Nichols Jr, William R. “The Cost and Benefits of Static Analysis during Development.” ArXiv:2003.03001 [Cs], 5 Mar. 2020

34. Paliouras, Georgios. Scalability of machine learning algorithms. Diss. University of Manchester, 1993.

35. Heo, Kihong, et al. “Machine-Learning-Guided Selectively Unsound Static Analysis.” IEEE Xplore, 1 May 2017

36. Lee, Junhee, et al. “NPEX.” Proceedings of the 44th International Conference on Software Engineering, 21 May 2022

37. Gupta, Rahul, et al. “DeepFix: Fixing Common c Language Errors by Deep Learning.” Proceedings of the AAAI Conference on Artificial Intelligence, vol. 31, no. 1, 12 Feb. 2017.

38. Long, Fan, and Martin Rinard. “Automatic Patch Generation by Learning Correct Code.” Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016, 2016

39. Mashhadi, Ehsan, and Hadi Hemmati. “Applying CodeBERT for Automated Program Repair of Java Simple Bugs.” ArXiv:2103.11626 [Cs], 30 Mar. 2021

40. Chappelly, Timothy, et al. “Machine Learning for Finding Bugs: An Initial Report.” IEEE Xplore, 1 Feb. 2017

41. Gupta, Nitin, et al. “Data Quality Toolkit: Automatic Assessment of Data Quality and Remediation for Machine Learning Datasets.” ArXiv:2108.05935 [Cs], 5 Sept. 2021, arxiv.org/abs/2108.05935. Accessed 11 Apr. 2023

42. Hittmeir, Markus, et al. “On the Utility of Synthetic Data.” Proceedings of the 14th International Conference on Availability, Reliability and Security, 26 Aug. 2019.

43. ROCHA, ANDERSON, et al. “HOW FAR DO WE GET USING MACHINE LEARNING BLACK-BOXES?” International Journal of Pattern Recognition and Artificial Intelligence, vol. 26, no. 02, Mar. 2012, p. 1261001

44. Cifuentes, C., and A. Fraboulet. “Intraprocedural Static Slicing of Binary Executables.” IEEE Xplore, 1 Oct. 1997, ieeexplore.ieee.org/abstract/document/5726949?casa_token=ae6
wlxgfA4AAAAA:GKZ30KmHwixKZjF0QsKy2rLbxp65itEFxjDMvxht3G_bI6_ZSxOQg6MFtJ3rA
Accessed 12 Apr. 2023.

45. Mitchell, Tom M. “Machine Learning and Data Mining.” Communications of the ACM, vol. 42, no. 11, 1 Nov. 1999, pp. 30–36, https://doi.org/10.1145/319382.319388.

46. Aaronson, Alderic. “Following You Home: The Ethics of Targeted Advertising — Linkr.” Www.linkreducation.com, 29 Nov. 2019,

47. Yang, Zhou, et al. “Stealthy Backdoor Attack for Code Models.” ArXiv:2301.02496 [Cs]

12 Figures Bibliography

The figures not included in this bibliography were created for this report.

- Figure 1: SVM
  DataFlair Team. "Support Vector Machines Tutorial - Learn to Implement SVM in Python - DataFlair." DataFlair, 4 Aug. 2017

- Figure 2: Kernel Trick
  "Support Vector Machine (SVM) — Note of Thi." Dinhanhthi.com, dinhanhthi.com/support-vector-machine/.

- Figure 3: Feed Forward Neural Net Structure
  Terry-Jack, Mohammed. "Deep Learning: Feed Forward Neural Networks (FFNNs)." Medium, 3 May 2019

- Figure 4: One Hot Encoding
  Yedla, Anurag, et al. "Predictive Modeling for Occupational Safety Outcomes and Days Away from Work Analysis in Mining Operations." International Journal of Environmental Research and Public Health, vol. 17, no. 19, 27 Sept. 2020,

- Figure 7: ROC Curve
  Wikipedia Contributors. "Receiver Operating Characteristic." Wikipedia, Wikimedia Foundation, 20 Mar. 2019

13 Appendix

13.1 Machine Learning Models Hyper-Parameters

13.1.1 MLP model, Entire File and Functioned Data Training

Sklearn's MLPClassifier algorithm was used, all other hyperparameters were equal to their default value **Network Configuration:** 1 layer of 100 units
Maximum Iterations: 500

13.1.2 XGBoost model Entire File and Functioned Data Training

Sklearn's XGBClassifier algorithm was used, all hyperparameters were equal to their default value

13.1.3 SVM model in Entire File and Functioned Data Training

Sklearn's svm algorithm was used, all other hyperparameters were equal to their default value
kernel: 'linear' **C:** 1

13.1.4 Hybrid model in Entrie File and Functioned Data Training

Sklearn's MLPClassifier algorithm was used, all other hyperparameters were equal to their default value **Network Configuration:** 3 layers of 150, 75,50 units
Maximum Iterations: 500

Sklearn's XGBClassifier algorithm was used, all other hyperparameters were equal to their default value **objective:** 'binary:logistic'
eval_metric: 'logloss'
'max_depth': 5
'eta': 0.1
'subsample': 0.8
'colsample_bytree': 0.8
'seed': 42