Displaying Explainability and Testing Robustness of Machine Learning Models in Software Engineering

Edward Passagi
University of Illinois, Urbana-Champaign
Illinois, USA
passagi2@illinois.edu

Simon Tonov
University of Illinois, Urbana-Champaign
Illinois, USA
stonov2@illinois.edu

ABSTRACT

Software developers spend most of their time debugging their code, which translates to decreased productivity. Tools that can aid with debugging can thus help developers spend less of their time on debugging and more on adding new features or being productive in other ways. Creating such tools, however, is not trivial. Finding similar code snippets and bug fixing are tasks that require analytical and comparative thinking, which are properties that cannot be programmed easily by a human. However, recent work in Machine Learning (ML) has shown that, instead of being programmed manually, software development tools can utilize natural language models that can learn analytical and comparative thinking.

In this work, we examine the explainability and robustness of three such models on a specific task, namely CodeT5[1], SequenceR[2], and GraphCodeBERT[3]. We propose methods that show whether the models are learning from relevant features (*explainability*) and potential adversarial attacks that target assumptions the models make or weaknesses in their implementation.

1 INTRODUCTION

In this proposal, we evaluate three neural models for software engineering aid on their specific sub task. We focus on the following:

(1) Clone Detection with CodeT5
(2) Bug fix with SequenceR
(3) Clone Detection with GraphCodeBERT

The following section provides brief explanations for each of the models on how it works, how it achieves the aforementioned sub-task, and the dataset that its using for training and evaluation.

2 MODEL, TASK, DATASET

2.1 CodeT5

CodeT5 builds on an encoder-decoder framework with the same architecture as T5[4], where T5 itself closely follows its originally proposed form[5]. CodeT5 neural architecture takes in either Programming (PL)-only or Natural Language(NL)-PL as its inputs, depending on whether the code snippet has accompanying NL descriptions. The model initially encodes the NL and PL inputs, focusing on the type of identifiers (e.g. function and variable names) from the PL, since they reserve rich code semantics. This processed input is then further analyzed during pre-training to extract useful patterns, which includes Identifier-aware Denoising Pre-Training, Identifier Tagging, Masked Identifier Prediction, and Bimodal dual generation.

In the clone detection task, CodeT5 first obtains the sequence embedding of each code snippet using the last decoder state (similar to BART's[6] approach). The final clone prediction is then calculated by measuring the similarity between the two decoder states of the input snippets.

The model is pre-trained on CodeSearchNet[7] which can be accessed here. Google Code Jam (GCJ)[8] and the widely-available BigCloneBench[9] datasets of the Java programming language are used to measure CodeT5's performance on code clone detection.

2.2 SequenceR

SequenceR is a sequence-to-sequence deep learning model designed to generate single-line fixes in buggy code in the *Bug Repair* task. More specifically, SequenceR takes in a buggy line and its surrounding context (*abstract buggy context*) as input to generate a bug fix prediction. To do this, SequenceR uses an encoder-decoder neural architecture that learns how to encode the abstract buggy context, how to produce bug fixes, which encoding values to pay attention to, and how to choose substitutions.

The encoder's purpose is to learn how to encode the *abstract buggy context* into a vector that is easily interpretable by other parts of the program. After the abstract buggy context has been encoded, an attention mechanism is used to generate a more specific context vector c_j, which allows the decoder to focus on certain values from the encoding while calculating y_j, the token to generate.

Chen et al.[2] merge the CodRep[10] and Bugs2Fix[11] datasets to create a dataset to train SequenceR on. To ensure the two datasets are in the same format, Chen et al.[2] extract only the one-line commits from Bugs2Fix.

2.3 GraphCodeBERT

GraphCodeBERT is a graph-based pre-trained model based on Transformers[5] for programming languages[3]. GraphCodeBERT utilizes data flow, which is a graph that represents the dependency relations between variables. The data flow provides a way to abstract the structure of a code snippet. An example of data flow can be seen in Figure 1. The model takes source code paired with a comment and the generated data flow as the input. It then runs the pre-training tasks which consist of Masked Language Modeling, Edge Prediction, and Node Alignment. Figure 1 shows the full architecture of the model.

In the clone detection task, the probability of two code snippets being true clones is calculated by dot product from the representation of [CLS]. That is, given two code snippets as an input, GraphCodeBERT will evaluate these two scripts and compare their resulting [CLS] representation to determine its final prediction.

GraphCodeBERT uses the dataset provided by Wang et al.[12] BigCloneBench[9] for training/validation/testing.

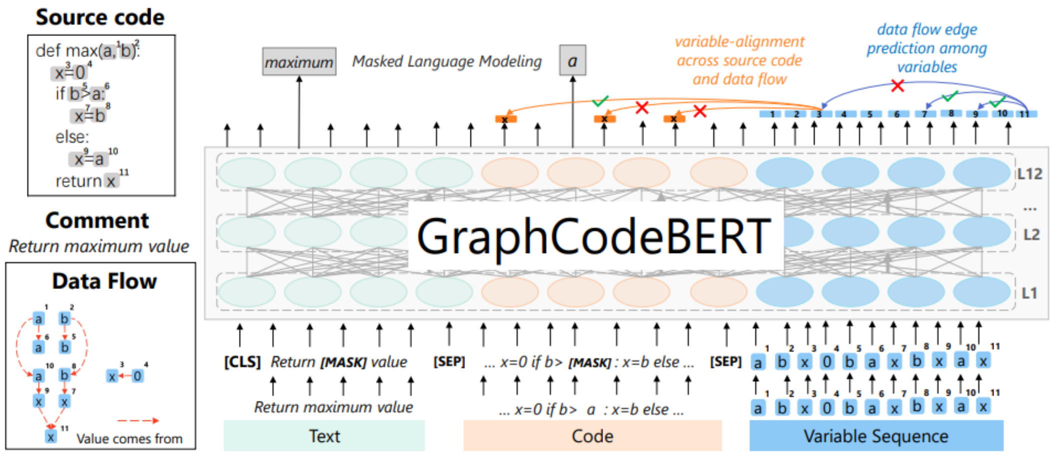

Figure 1: GraphCodeBERT architecture [3]

3 INTERPRETATION TECHNIQUE

3.1 CodeT5

CodeT5 is a pre-trained encoder-decoder model that considers the token type information in code. The CodeT5 model is built upon the T5 architecture (Raffel et al 2020).

3.1.1 Hypothesis. The code clone detection task aims to measure the similarity between two code snippets and predict whether they have the same functionality. CodeT5 achieves this by measuring the similarity between the last decoder states (sequence embeddings) of two code snippets. Since the decoder state is affected by the encoder, which takes in the programming language (PL) and natural language (NL) tokens as its input, the model's ability in detecting code clones should heavily be affected by the context (tokens) that surround each PL and NL (if available) token.

Architecturally, CodeT5 is the same as T5 (Figure 2). Each of its decoder component takes in the encoders' output alongside the previous decoder result (or start condition) when configuring its encoder-decoder attention. The encoder itself considers the context surrounding each of the token when calculating the score vector for each token. This computation will eventually determine the sequence embedding of the last decoder, which is what CodeT5 evaluates when determining code similarities. Since CodeT5 provides either PL-NL for bimodal inputs or PL-only for unimodal inputs, its ability to detect clones will be heavily dependent on contextual similarities of each of these input tokens.

3.1.2 Proposed Algorithm. Our algorithm will take two code snippets (i.e. source_code1, source_code2) to determine its similarities through CodeT5. Our proposed algorithm will follow CodeT5's method of calculating code similarity, with some additional steps to keep track of the relevant states and information between each of the computation process. That is, for each source code, we will:

(1) Generate the PL-only (for unimodal) or NL-PL (for bimodal) tokens and form it as a proper input for CodeT5's encoding pre-training step.
(2) Run it through the encoding step, keeping track of the final score vector of each token.
(3) Run it through the decoding step, keeping track of the sequence embedding values within each decoder step until it reaches the stopping condition.

Once the decoder is finished, following CodeT5's clone detection implementation, we can then compare the similarities between the last decoder state of each source code to determine the prediction. We'll also store all of the scores of the transitional steps of each source code as a raw declarative representation [14].

Our proposed algorithm form the output as (final_prediction, representation1, representation2), where representation represents the raw declarative representations of source_code1 and source_code2 respectively.

3.1.3 Illustrative Example. Given two source codes to be evaluated, our algorithm will return the final CodeT5 prediction alongside the raw declarative representations of each of the source code. We can then evaluate this representation to determine how the initial output of the encoding step affect the transitional sequence embeddings of the decoders, and ultimately, its final decoder state.

In particular, we can evaluate how the context that surrounds each of the PL and NL tokens translate to the score vector generated by the encoder, and see how this affects the final prediction calculated by CodeT5. This step will allow us to verify our hypothesis that for code generation, CodeT5 is heavily dependent on the context that surrounds each of the PL and NL tokens.

3.2 SequenceR

3.2.1 Hypothesis. In the bug repair task, SequenceR's context vector should place high weights on the segment of the encoding corresponding to the buggy line in the abstract buggy context. The context vector should also place high weights on all other occurrences of identifiers (or functions) referenced in the buggy line and their immediate context.

When developing a solution to a single-line bug, software developers pay most attention to the identifiers and function calls in that

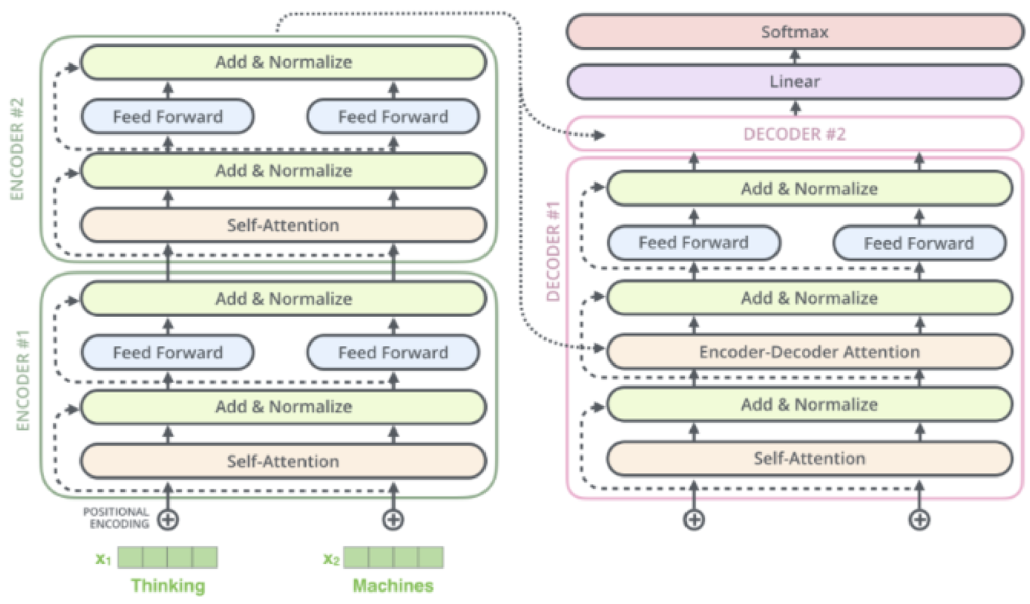

Figure 2: Generic T5 architecture[13]

line. The debugging process involves searching for other occurrences of any of the identifiers in the buggy line and examining the context that surrounds them. Eventually, by examining the context around other occurrences of the identifiers from the buggy line, we might find an example of how they are properly used, giving us insight as to how we can fix our buggy line.

SequenceR uses an attention mechanism that allows it to focus on certain parts of the encoding of the abstract buggy context, mimicking the way human programmers focus on relevant parts of the program. The attention mechanism calculates weights for the various parts of the abstract buggy context and outputs a context vector, so we conjecture that the segments of code that a human is most likely to focus on should contain the most relevant features for learning; as such, these segments of the code should have the highest weights in the resulting context vector.

3.2.2 Proposed Algorithm. We are interested in producing a saliency map for each token in the bug fix to verify if SequenceR has been learning from what we suppose are the most relevant features. Since SequenceR does not produce such explainability visualizations out-of-the-box, we need to construct a white-box algorithm that uses a modified version of SequenceR.

Our modified version of SequenceR will take a buggy code snippet that follows the expected input format (an entire buggy class, a selected buggy method, and a selected buggy line) and process this code snippet just like the original SequenceR algorithm with one addition; for each output token y_j generated by our algorithm, we will use the context vector c_j produced by the attention mechanism to produce a saliency map. In the end, we will have the fixed line just like the original context, but for each of its tokens, we will also have a saliency map justification.

To produce a saliency map for a token y_j, we need its corresponding context vector and the abstract buggy context. However, we cannot compare them directly. Since the context vector c_j was calculated using the encoding of the abstract buggy context, we want to reverse the mapping done by the encoder and apply the resulting operation to the context vector to see how the weights get mapped to words. To accomplish this, we use the gradients of the encoding step (though without modifying any values) and propagate the context vector c_j in reverse through the LSTM layers of the encoder until we reach the unencoded abstract buggy context.

Having the distribution of weights across the abstract buggy context, creating the saliency map involves printing the abstract buggy context with the tokens highlighted in various shades corresponding to the strength of the weights for that particular location. This saliency map is considered in the output for a particular token y_j generated by the decoder.

3.2.3 Illustrative Example. Suppose we have a buggy code that consists of a class with several methods, one of which is buggy and contains a buggy line we want to fix. We pass this buggy code into our modified SequenceR algorithm which gives us a suggested fixed line and a list of saliency maps, one for each token. We are now free to examine the saliency map for each token and see whether SequenceR learned from the relevant features.

3.3 GraphCodeBERT

3.3.1 Hypothesis. GraphCodeBERT neural model considers both the data flow and the syntax of a source code. In the clone detection task, syntactically, the model should consider operators above identifiers, while placing minimal weights to other types of tokens. We should expect the variable sequence to have a similar weight as the source code, since they are both crucial in determining code similarities.

Intuitively, the operators are the elements of a code segment that dictates how an output is generated. Swapping an operator with another one will likely change the output of the code, so having operator similarity between two code segments suggests similarity of function. On the other hand, the names attached to identifiers do not influence the output. If we swap the name of all instances of one identifier with an unseen name, the output would not change. However, programmers tend to attach meanings to identifiers to indicate the purpose of a variable. Thus, having identifier similarity could also suggest a similarity of function between two code segments. Lastly, all other tokens (e.g. semicolon, parentheses, commas, etc.) do not intuitively carry any meaning about the purpose of a program. That is, they are simply a syntactic property of a specific programming language.

As the paper describes it, the data flow graph is a data structure that represents the dependency relations between variables; for any given variable, which other variables does its value depend on. Similarities in the data flow between two code segments can be a strong indication of code clones. The paper demonstrates this by comparing two code segments that produce different outputs with virtually the same syntax. The data flow-blind approach labeled the segments as code clones, but the approach that considers data flow correctly identified them as different. However, the data flow graph does not consider the operations between the variables; this information is carried by the source code. Thus, we expect the variable sequence—the subcomponent of the data flow graph that GraphCodeBERT's neural model uses—to have similar weights as the source code.

3.3.2 Proposed Algorithm. To verify our hypothesis, a saliency map would prove useful. We propose a white-box approach that uses the results of each of GraphCodeBERT's layers to compute a saliency map of the most important features for each of the code snippets.

As an input, GraphCodeBERT's clone detection task takes two code snippets and outputs 1 or 0, depending on whether the snippets are clones or not. Our proposed white-box approach also takes in two code snippets, but in addition to outputting 1 or 0, it also outputs two saliency maps for each of the code snippets. These saliency maps are displayed on top of the source code snippets and represent which tokens from the source codes were the most significant in the embedding.

To do this, we employ a first-derivative saliency that estimates how much one layer influenced a subsequent layer. This will help us figure out which features of the source code were most pronounced in the embedding. After getting the two saliency maps for the source codes, we manually examine if GraphCodeBERT really paid attention to what we hypothesized were the most important features.

3.3.3 Illustrative Example. First-derivative saliency maps are effective in displaying the most important features that contributed to an output. With such a visualization, we can easily inspect what the model most paid attention to in pairs of clones, and whether its reasoning is correct. For example, let GraphCodeBERT correctly predict two source codes to be clones. After examining the two saliency maps, however, suppose we find that GraphCodeBERT gave high values to semicolons and low values to the data flow graph. This suggests that GraphCodeBERT labeled the two source codes as clones mostly because they had the same number of semicolons and not because of any possible similarities in the data flow graph. Instead, we should expect GraphCodeBERT to be paying most attention to the operators and identifiers of the two code snippets while also taking the data flow graph into account; such an observation would imply that GraphCodeBERT is learning from relevant features.

4 ADVERSARIAL ATTACKS

4.1 CodeT5

CodeT5 uses Google Code Jam (GCJ) and BigCloneBench for training and evaluating their model. Given this benign snippet from BigCloneBench:

```
if (a>=b) {
   c=d+b;
   d=d+1;
} else
   c=d-a;
```

Here are some potential adversarial attacks toward CodeT5:

(1) *Redundant Variable Flooding.* Generate and add a bunch of code that does nothing (i.e. variable declarations on non-existent variables within the script). This will introduce many useless PL tokens to obfuscate the context. This white-box attack can be run during training time (reducing the model's learning ability) or evaluation time (reducing the model's ability to predict correctly). This attack will transform our benign snippet into:

```
z0=0;
z1=0;
z0=0;
if (a>=b) {
   c=d+b;
   d=d+1;
   z1=1;
} else
   c=d-a;
   z0=0;
```

(2) *Operator Replacements.* Utilize external libraries to replace common operators with its similar but syntactically different function. This will not change the final code outputs. This will cause our model to not be able to evaluate similarity through operators' resemblance, which will reduce its prediction's ability. Our input will be transformed into:

```
if (a>=b) {
   c=addition(d,b);
   d=addition(d,1);
} else
   c=subtraction(d,a);
```

(3) *Non-descriptive Identifier Names.* Since CodeT5 assumes that developers tend to use descriptive names, this attack replaces all identifier names with non-descriptive ones. This

white-box attack can run during training time (i.e. the model can't learn from descriptive identifier names) or testing time (i.e. the model can't infer from previously learned descriptive names). On input with non-descriptive variable names, we can switch the variable names around to further disable the model's ability in learning the pattern behind identifier names. Our benign input will be transformed into:

```
if (p>=q) {
    r=s+q;
    s=s+1;
} else
    r=s-p;
```

(4) *Garbage Descriptive Identifier Names.* Use randomized, descriptive identifier names. This white-box attack replaces all the variable within the script to prevent the model from identifying useful patterns behind a descriptive identifier names (i.e. totalSum does not neccessarily imply summation of a final total value). This attack can be run during learning time (preventing the model from learning useful descriptive patterns) or training time (preventing the model from inferring descriptive meanings). This attack will transform our benign snippet into:

```
// replacing: a->totalSum, b->totalProd,
     c->curTime, d->timeElapsed
if (totalSum>=totalProd) {
    curTime=timeElapsed+totalProd;
    timeElapsed=timeElapsed+1;
} else
    curTime=d-totalSum;
```

(5) *Duplicate Extension.* Append the same no-op chunk onto the end of both the input when evaluating the similarity between two codes. This black-box attack that runs during testing time will confuse the model into thinking that the two inputs are the same due to their shared syntactic similarities from the appended block. This attack will transform our benign snippet into:

```
if (a>=b) {
c=d+b;
d=d+1;
} else
    c=d-a;
// program ends

// this appended arbitrary code segment does
     nothing
int y=0;
int z=1;
y=y+z;
z=z+y;
y=y+y;
z=z+z;
```

4.2 SequenceR

SequenceR uses a combination of the CodRep[10] and Bugs2Fix[11] datasets to train and test their model. They also add <START_BUG> and <END_BUG> to delineate the bug. A sample buggy code (*abstract buggy context*) is shown below:

```
public class InvalidMatrixExceptionTest extends TestCase {
    public void testConstructorMessage() {
        String msg = "message";
        <START_BUG>
        InvalidMatrixException ex = new
             InvalidMatrixException(msg, new Object[0]);
        <END_BUG>
        assertEquals(msg, ex.getMessage());
        }
    }
```

We have developed five attacks to thwart the performance of SequenceR:

(1) *Vocabulary Flooding.* Occasionally, SequenceR uses a copy mechanism to replace identifiers with other identifiers that appear in the abstract buggy context. This happens whenever SequenceR calculates that it needs to copy a token that is not from the vocabulary V. As a reminder, V is a vocabulary of the 1,000 most common tokens.
Let's assume that sequencer decided to use the copy mechanism, i.e. choose words *outsude the vocabulary*. We can trick SequenceR by flooding the buggy code with tokens from the vocabulary; the whole intuition of the copy mechanism is for SequenceR to choose *out-of-vocabulary* words, so with this attack, we are testing if SequenceR will avoid vocabulary words when they appear in the *abstract buggy context*. Impairing the copy mechanism impairs SequenceR's performance because, in many cases, the bug fix involves *out-of-vocabulary* words. To do this attack, we append many no-op declarations that use tokens from the vocabulary; #COMMON_IDENTIFIER_K is a random common identifier from the vocabulary.

```
public class InvalidMatrixExceptionTest extends
     TestCase {
public void testConstructorMessage() {
    String #COMMON_IDENTIFIER_0 = null;
    String #COMMON_IDENTIFIER_1 = null;
    ...
    String #COMMON_IDENTIFIER_N = null;
    String msg = "message";
    <START_BUG>
    InvalidMatrixException ex = new
         InvalidMatrixException(msg, new
         Object[0]);
    <END_BUG>
    assertEquals(msg, ex.getMessage());
    }
}
```

(2) *Buggy Line Flooding.* We can flood the *abstract buggy context* with the buggy line in question to trick SequenceR into learning from the buggy line rather than other code. This way, SequenceR will be less likely to change the buggy line into a

fixed line. We can accomplish this by repeating the segment between the <START_BUG> and <END_BUG> tokens many times. This modification can easily cause the buggy code to have a different output or to not compile; the buggy code below being unaffected is an exception.

```
public class InvalidMatrixExceptionTest extends
     TestCase {
public void testConstructorMessage() {
    String msg = "message";
    InvalidMatrixException ex = new
         InvalidMatrixException(msg, new
         Object[0]);
    ...
    <START_BUG>
    InvalidMatrixException ex = new
         InvalidMatrixException(msg, new
         Object[0]);
    <END_BUG>
    assertEquals(msg, ex.getMessage());
    }
}
```

(3) *Declared Identifier Substitution.* The bug fix should not rely on the names chosen for the identifiers declared in the buggy code. For example, whether the developer chose "msg" or "message", the bug/bug fix should remain the same. This adversarial attack checks if SequenceR is learning from these names, which we think should not be the case. For each identifier declared in the buggy code, we can replace all occurrences with a different, unused identifier. After we get the output of SequenceR, we revert the identifiers of the fixed line to their original names to see if the bug fix matches.

```
public class InvalidMatrixExceptionTest extends
     TestCase {
public void testConstructorMessage() {
    String i000 = "message";
    <START_BUG>
    InvalidMatrixException i001 = new
         InvalidMatrixException(i000, new
         Object[0]);
    <END_BUG>
    assertEquals(i000, i001.getMessage());
    }
}
```

(4) *No-op Identifier Flooding.* The intuition behind this attack is to test whether SequenceR learns from code that is not connected logically to the bug. Here, we flood the buggy code with many declarations of unique identifiers that are in no way connected to the existing code. SequenceR should not learn from these identifiers.

```
public class InvalidMatrixExceptionTest extends
     TestCase {
public void testConstructorMessage() {
    String i000 = null;
    String i001 = null;
    ...
    String msg = "message";
    <START_BUG>
    InvalidMatrixException ex = new
         InvalidMatrixException(msg, new
         Object[0]);
    <END_BUG>
    assertEquals(msg, ex.getMessage());
    }
}
```

(5) *Cheat Code.* Chen et al.[2] found that SequenceR is able to predict the bug fix in 950/4,711 of the prediction tasks. For the tasks that it failed, we want to test if SequenceR can find the correct prediction if we include the fixed line somewhere in the code. This attack tests whether SequenceR can learn from correct implementation to fix the buggy line. If SequenceR's performance does not improve, this would suggest that the model overlooks important clues when developing the fix. We modify the buggy code by inserting the fixed line close to the buggy line. To ensure the output of the code snippet is the same, we can simply preface the fixed line with a conditional that never executes.

```
public class InvalidMatrixExceptionTest extends
     TestCase {
public void testConstructorMessage() {
    String msg = "message";
    if (false) {
       InvalidMatrixException ex = new
            InvalidMatrixException ( msg , null
            ) ;
    }
    <START_BUG>
    InvalidMatrixException ex = new
         InvalidMatrixException(msg, new
         Object[0]);
    <END_BUG>
    assertEquals(msg, ex.getMessage());
    }
}
```

4.3 GraphCodeBERT

GraphCodeBERT uses the same datasets as CodeT5 (GCJ[8] and BigCloneBench[9]). Since they're both evaluating code clones, some attacks can be shared, though we provide explanation on which vulnerability this attack exploits within GraphCodeBERT. We're using the same benign snippet as the one declared in CodeT5 (Section 4.1). Here are some potential adversarial attacks toward CodeT5:

(1) *Redundant Variable Arithmetic.* This white-box attack exploits GraphCodeBERT's data flow evaluation. By doing redundant variable arithmetic, the data flow will contain unnecessary vertices toward these redundant variables, reducing its ability to detect clones. This attack can be done during training time (i.e reducing the model's ability to learn the data flow) or evaluation time (i.e. reducing the model's ability of inferring the data flow correctly). Our input will be transformed into:

```
int z=0;
int y=0;
if (a>=b) {
    c=d+b+z+y;
    d=d+1+z+y;
} else
    c=d-a+z+y;
```

(2) *Circular Dependency.* On the topic of data flow exploitation, this white-box attack focuses on generating circular dependency within the data flow to decrease the model's ability from meaningfully learn the snippet's data flow. This attack runs during evaluation, modifying the two given snippets from the input. This attack will cause the model to catch similarities between two data flows, driving the model to almost always predict duplicates even in two very different scripts. This attack will transform our input into:

```
int z=0;
int y=0;
for (int i=0; i<99; i++) {
    z=y;
    y=z;
}
if (a>=b) {
    c=d+b;
    d=d+1;
} else
    c=d-a;
```

(3) *Redundant Variable Flooding.* As previously described in attack 1 of CodeT5. This attack will affect both the syntactic and data flow evaluation of GraphCodeBERT.

(4) *Operator Replacements.* As previously described in attack 2 of CodeT5. This attack will only affect the syntactic evaluation of GraphCodeBERT.

(5) *Duplicate Extension.* As previously described in attack 5 of CodeT5. This attack will affect both the syntactic and data flow evaluation of GraphCodeBERT.

5 CONCLUSION

In this proposal, we examined three algorithms to check whether each of the chosen models is learning from relevant features. In two of our algorithms, we chose saliency maps[14] to visualize the particular model's justification, and for CodeT5, we observed the raw declarative representations[14] of the model's inputs. These algorithms can be used to show whether the model in question is learning from the relevant features, giving insight to the model's developers as to what can be improved.

We also proposed adversarial attacks that modified the dataset to confuse each model into giving an incorrect prediction or to test whether the model is performing up to our expectations. They try to tackle assumptions made by the developers about the nature of source code, so that they can improve their models and their quality-of-life.

REFERENCES

[1] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021. URL https://arxiv.org/abs/2109.00859.

[2] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. SEQUENCER: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, pages 1–1, 2021. doi: 10.1109/tse.2019.2940179. URL https://doi.org/10.1109%2Ftse.2019.2940179.

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. URL https://arxiv.org/abs/2002.08155.

[4] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019. URL https://arxiv.org/abs/1910.10683.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

[6] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.703. URL https://aclanthology.org/2020.acl-main.703.

[7] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019. URL https://arxiv.org/abs/1909.09436.

[8] Code jam. URL https://codingcompetitions.withgoogle.com/codejam.

[9] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, 2014. doi: 10.1109/ICSME.2014.77.

[10] Zimin Chen and Martin Monperrus. The codrep machine learning on source code competition, 2018. URL https://arxiv.org/abs/1807.03200.

[11] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation, 2018. URL https://arxiv.org/abs/1812.08693.

[12] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural networkand flow-augmented abstract syntax tree, 2020. URL https://arxiv.org/abs/2002.08653.

[13] Qiurui Chen. T5: A detailed explanation, Jun 2020. URL https://medium.com/analytics-vidhya/t5-a-detailed-explanation-a0ac9bc53e51.

[14] Marina Danilevsky, Kun Qian, Ranit Aharonov, Yannis Katsis, Ban Kawas, and Prithviraj Sen. A survey of the state of explainable ai for natural language processing. 2020. doi: 10.48550/ARXIV.2010.00711. URL https://arxiv.org/abs/2010.00711.