Interpreting CodeBERT for Semantic Code Clone Detection

Shamsa Abid, Xuemeng Cai, Lingxiao Jiang
School of Computing and Information Systems
Singapore Management University
{shamsaabid, xuemengcai, lxjiang}@smu.edu.sg

***Abstract*—Accurate detection of semantic code clones has many applications in software engineering but is challenging because of lexical, syntactic, or structural dissimilarities in code. CodeBERT, a popular deep neural network based pre-trained code model, can detect code clones with a high accuracy. However, its performance on unseen data is reported to be lower. A challenge is to interpret CodeBERT's clone detection behavior and isolate the causes of mispredictions. In this paper, we evaluate CodeBERT and interpret its clone detection behavior on the SemanticCloneBench dataset focusing on Java and Python clone pairs. We introduce the use of a black-box model interpretation technique, SHAP, to identify the core features of code that CodeBERT pays attention to for clone prediction. We first perform a manual similarity analysis over a sample of clone pairs to revise clone labels and to assign labels to statements indicating their contribution to core functionality. We then evaluate the correlation between the human and model's interpretation of core features of code as a measure of CodeBERT's trustworthiness. We observe only a weak correlation. Finally, we present examples on how to identify causes of mispredictions for CodeBERT. Our technique can help researchers to assess and fine-tune their models' performance.**

***Index Terms*—Explainable AI, Model Interpretation, Black-box, Semantic Clone Detection, Code Model, Deep Learning**

I. Introduction

Code clones support software engineering tasks such as program understanding, aspect mining, copyright infringement investigation, and software evolution analysis; making clone detection important for various software analysis tasks [1]. Semantic clones present a challenge for most code clone detection tools since semantic clones may not be textually similar. Several semantic code clone detection techniques have been proposed in the recent years [2–13] with a growing interest in applying machine learning techniques. These deep learning-based code models report high accuracy for the clone detection task, however, interpretability of the code models remains to be explored. These deep learning-based models are intrinsically complex with their multilayered architectures, and it is unclear how and why these models make the predictions. For clone detection in particular, any instances of misprediction are a red flag for the quality of the model or data and are worthy of investigation.

CodeBERT is a popular deep neural network based pre-trained code model which is reported to detect code clones with a high accuracy (94% F1-score) [13, 14]. However, recent studies show that the performance of CodeBERT for code clone detection decreases, (27-47% drop in F1-score [15] and 15-40% drop in recall [16]), when evaluating different code than those used for model building. However, there is no explanation study that attempts to discover the reasons why the model mispredicts on unseen data. We need a solution to interpret CodeBERT's predictions for clone detection to get insight into the decision-making behavior of CodeBERT and to identify the causes of mispredictions. Furthermore, we need to measure the level of confidence we can have in CodeBERT's predictions for detecting code clones.

The key research questions of our study are thus as follows: *What parts of code in a clone pair is CodeBERT looking at to classify it as a semantic clone? How far or close the model's code comprehension behavior is to human's code comprehension? How can we identify causes of mispredictions?*

In this paper, we aim to explain CodeBERT's [14] clone detection predictions for semantic code clones. We use CodeBERT to classify a set of 115 Java and 120 Python clone pairs[1] from the SemanticCloneBench [17] dataset and then use a black-box model interpretation technique to *explain the behavior of the model*. First, we manually determine the semantic similarity between methods in a clone pair and assign clone and non-clone labels. This labeling helps identify mislabeled clone pairs for a fair evaluation of the model. We then calculate CodeBERT's accuracy of predicting semantic clone pairs using our updated clone labels. We also manually assign importance labels to each method's statements to quantify their contribution to the *core* functionality as per human understanding. Second, we use SHAP [18], an attribution-based interpretation technique, to obtain the importance scores of individual statements (including header[2]) of each method in a clone pair as a measure of their contribution towards the clone prediction. We refer to our application of SHAP to clones pairs as *CloneSHAP*. Using the SHAP values, we then identify the key statement types of a method that Code-

This research is supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

[1]Clone pairs include both true and false pairs unless specified otherwise.

[2]Whenever we refer to code statements of a method it includes the method's header as well.

BERT pays attention to for the clone label prediction. This provides insight into the key statements influencing a model's prediction. Third, we evaluate the correspondence between human and machine interpretation of important statements. Our intuition is as follows: If statements with higher SHAP values also have higher importance levels assigned by humans, then we may infer that the model's prediction is based on core functionality similar to human intuition. This, in turn, establishes the model's trustworthiness. Finally, we present some examples on how to identify the parts of code causing the mispredictions for CodeBERT using our interpretation method based on SHAP values.

We make the following main contributions:

- We perform a manual analysis of 115 Java and 120 Python clone pairs from SemanticCloneBench to assign a binary similarity label to clone pairs. Our manual labeling of clone pairs similarity identifies 49 mislabeled Java clone pairs and 42 mislabeled Python clone pairs. Using our manually assigned clone labels as ground truth, we evaluate CodeBERT's semantic clone prediction accuracy as 56.5% on Java and 54.45% on Python clone pairs.
- We manually assign importance scores to Java statements and Python statements constituting the 115 Java and 120 Python clone pairs as part of creating a dataset of human-intuitions for code statements that reflect the *core* functionality of a method.
- We use SHAP values to identify key statements of methods in a clone pair that contribute towards a prediction. *Method headers* and *for statements* have the highest average impact on CodeBERT's true predictions and false predictions respectively for Java clone pairs. *Return statements* have the highest average impact on CodeBERT's true and false predictions for Python clone pairs.
- Using the SHAP values, we discover that machine-interpretation of statements correlates with human-intuition only as much as 53.2% of the time for Java and 58.33% of the time for Python clone pairs. Thus, we can infer that CodeBERT does not analyze code the same way as humans do. Because of this weak correlation, our confidence in CodeBERT for semantic clone detection is also weak.
- We present some mispredicted cases of Java and Python clone pairs, and demonstrate how to identify the cause of misprediction using SHAP values.

To support the replication of our experiment and results, we have uploaded our code, our labeled data, and the results of our experiments online on GitHub [19].

II. Background

A. Semantic Code clones

Semantic code clones are also known as Type-4 clones. These clones include code fragments that are semantically similar in terms of functionality, but possibly different in how the functionality is implemented. These types of clones may have little or no lexical or syntactic similarity and hence are difficult to detect. Figure 1 shows an example of a semantic code clone pair found in SemanticCloneBench which is correctly identified by CodeBERT.

```
def is_sorted(stuff) :
 for i in range(1, len(stuff)) :
  if stuff [i - 1] > stuff [i] :
   return False
 return True
```

Listing 1: Clone208.py - A

```
def is_sorted(stuff) :
 for index, item in enumerate(stuff) :
  try :
   if item > stuff [index + 1] :
    return False
  except IndexError :
   return True
```

Listing 2: Clone208.py - B

Fig. 1: Python clone pair which CodeBERT correctly detects as a semantic clone.

The constituent clone pairs of SemanticCloneBench are sourced from code snippets found in answers to questions on Stack Overflow [20]. The general intuition is that code solutions to the same question will be semantically similar. StackOverflow questions may ask for a specific code example to implement a functionality, ask for bug fixes, or even ask for explanations of program behavior. However, some Stack Overflow questions may ask for functionality for which there may be semantically dissimilar solutions. For example, for the question, "How can I create a memory leak in Java?", there may be solutions with entirely different functionalities to achieve the goal of a memory leak. We observe that such cases of Java clone pairs in the SemanticCloneBench dataset are not valid semantic clone pairs. In this paper, we identify the mislabeled clone pairs that we randomly sample from SemanticCloneBench.

B. CodeBERT

CodeBERT [14] is a pre-trained model for natural language (NL) and programming language (PL). It is a multi-layer bidirectional transformer and uses attention [21]. CodeBERT uses the same model architecture as that of RoBERTa-base [22]. RoBERTa is a successor of BERT with optimized pre-training. CodeBERT was trained with the settings used for training multilingual BERT [23]. The CodeBERT model has 125 million parameters. CodeBERT is trained on six programming languages Ruby, JavaScript, Go, Python, Java, and PHP. For our experiments, we use CodeBERT that is originally tuned on clone pairs from BigCloneBench [24].

To predict the semantic similarity using CodeBERT, we pass two methods to a tokenizer. [CLS] and [SEP] tokens are added at the start and end of a method's tokens and each method is padded with [PAD] tokens to get the same number of tokens for each method. The padded sequences are then concatenated and passed to the CodeBERT model. CodeBERT then predicts the label of the method pair as 1 if they are similar or 0 if they are dissimilar.

C. SHAP Theory

SHAP (SHapley Additive exPlanations) [18] is a game theoretic approach to explain the output of any machine learning model. Shapley values calculate the importance of a feature by comparing what a model predicts with and without the feature. However, since the order in which a model sees features can affect its predictions, this is done in every possible order, so that the features are fairly compared. SHAP explanation outputs a base value, an expected value, and feature contribution values of dataset features. The base value is an average model output over the training dataset. The expected value is the model output over a test sample. The feature contribution values are between -1 and +1. These values may either be positive or negative; pushing the model output from the base value to model output. Positive SHAP value means positive impact on prediction, leading the model to predict 1. Negative SHAP value means negative impact, leading the model to predict 0.

III. Interpretation Approach

For interpreting CodeBERT's semantic clone detection, we filter (subsection III-A, process (subsection III-A1), and select a subset of Java and Python clone pairs from SemanticCloneBench). We manually label our selected clone pairs and assign true or false clone labels depending on whether both methods of the clone pair are semantically similar or not (subsection III-B1). Having these clone labels as the ground truth allows us to perform a fair evaluation of CodeBERT's accuracy for detecting clone pairs. We also manually label individual statements based on human interpretation of contribution of a statement towards the core functionality of a method (subsection III-B2). We then execute our *CloneSHAP* interpretation technique (subsection III-C) on our sampled code pairs to get SHAP values of method statements within the code pairs. A correlation analysis between human-assigned statement labels and machine-interpreted SHAP values allows us to evaluate the CodeBERT model's trustworthiness. Furthermore, analyzing positive SHAP values for mispredicted code pairs leads us to identify statements causing the mispredictions.

A. Dataset Filtering

We filter out and select only those clone pairs that consist of at most 10 statements. The reason for this constraint is our lack of computation resources required to handle larger statement permutations for getting SHAP values of larger-sized methods.

1) Code Flattening: To minimize the syntactic violations of Java code while performing line-level mutations, we perform a *flattening* step such that statements spanning multiple lines appear on a single line. The flattening step also removes lines containing only curly braces which results in condensed Java code with each line of a method occupied by either its header or a statement. This will reduce the *CloneSHAP* Interpreter's processing overhead by reducing the number of statement permutations that need to be explored. To perform this *flattening*, we parse the AST of the Java code and extract only the method headers and certain statement types from the method's code. We extract statement types like variable declarations, expressions, *if* statements, *while* statements, *for* statements, and *return* statements. The interpretation technique operates only on the flattened Java code. Since Python code is free from curly brackets, it already has a condensed representation. We cannot remove any other lines to reduce processing overheads.

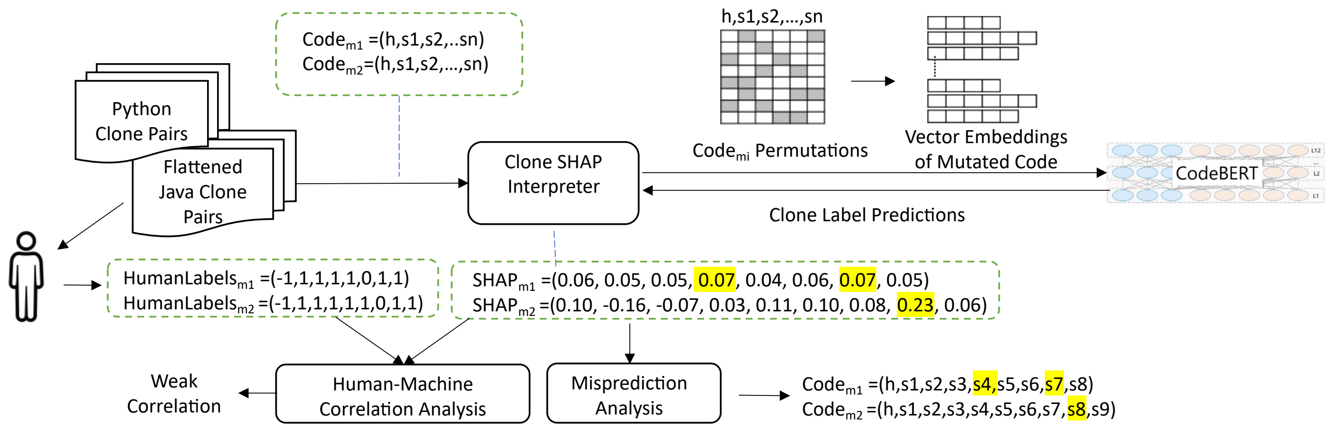
After filtering Java clone pairs with less than or equal to 10 statements, we obtain 230 Java clone pairs. For these clone pairs, we check whether reducing the methods in an original code file to our *flattened* format changes the original prediction. There are a total number of 115 code pairs where the *flattened* format preserves the original prediction. For the remaining 50% of the 230 code pairs, the model's prediction changes due to flattening either of the constituent methods. This indicates that the model's prediction is dependent on non-essential code parts 50% of the time. This is a key observation which implies that CodeBERT's predictions are unreliable. We exclude such clone pairs from our interpretation analysis and perform our experiments on only the 115 prediction-preserving flattened Java clone pairs.

After filtering Python clone pairs with less than or equal to 10 statements, we obtain 400 Python clone pairs. Out of these we sample 120 clone pairs. The margin of error for this sample size is 6% with a 90% confidence level for our results.

B. Human Intuition Labeling

Four human resources manually analyze the clone pairs' functionalities, assess the similarities, and assign true or false clone labels. Furthermore, they manually analyze and assign importance levels to statements of the sampled clone pairs. From the human evaluators, three are the authors of the paper, whereas the fourth is an external resource selected based on their development experience in Python. Two individuals H1 and H2 label the Java clone pairs and statements and have 7 years and 3 years of Java development experience and a third individual H3 who breaks the ties on clone pairs has 20 years of working experience with Java. H2 and another individual H4 label the Python clone pairs and have 4 and 4.5 years of Python development experience, while H3 who breaks the ties has 15 years of working experience with Python.

1) Labeling Clone Pairs: We manually evaluate the 115 Java and 120 Python clone pairs for semantic similarity. To determine the clone label for a clone pair, a human evaluator goes through the steps of reading both methods of a clone pair, referring to the StackOverflow user question (if needed), understanding the functionalities of the two methods, comparing similarities and differences, and finally assigning the clone label. For each clone pair, two human resources independently analyze the functionalities of both methods in a clone pair and assign a true label to a clone pair if they find both functionalities to be similar, and a false label if they find the methods to be dissimilar. To determine functional



Code$_{m1}$ =(h,s1,s2,..sn)
Code$_{m2}$=(h,s1,s2,...,sn)
h,s1,s2,...,sn
Python Clone Pairs
Flattened Java Clone Pairs
Clone SHAP Interpreter
Code$_{mi}$ Permutations
Vector Embeddings of Mutated Code
CodeBERT
Clone Label Predictions
HumanLabels$_{m1}$ =(-1,1,1,1,1,0,1,1)
HumanLabels$_{m2}$ =(-1,1,1,1,1,1,0,1,1)
SHAP$_{m1}$ =(0.06, 0.05, 0.05, 0.07, 0.04, 0.06, 0.07, 0.05)
SHAP$_{m2}$ =(0.10, -0.16, -0.07, 0.03, 0.11, 0.10, 0.08, 0.23, 0.06)
Weak Correlation
Human-Machine Correlation Analysis
Misprediction Analysis
Code$_{m1}$ =(h,s1,s2,s3,s4,s5,s6,s7,s8)
Code$_{m2}$ =(h,s1,s2,s3,s4,s5,s6,s7,s8,s9)

Fig. 2: Explaining the predictions of the CodeBERT model for semantic code clone detection

similarity, the evaluators focus on the presence of common functionality while ignoring any lexical, syntactic, and structural differences, as well as ignoring any additional code such as print statements and exception handling statements. A third human evaluator independently evaluates and assigns labels to only those clone pairs which were assigned different labels by the evaluators. We then use a majority vote to break any disagreements between the two evaluators. Since the clone pairs in the Java dataset include references to the StackOverflow question, the human evaluator uses the StackOverflow question and its context to get a better idea of the functionality of each method.

2) Labeling Statements: We refer to the statements in a method that are essential to the core functionality of a method as *core* statements. We refer to the statements which are not essential to the core functionality of the method as *non-core* statements. Finally, we refer to the statements for which the human evaluator can not make a firm decision regarding its relevance to the core functionality as *gray* statements. The two evaluators analyze each statement in a method and decide whether a statement is relevant to the core functionality of the method. The evaluator assigns a 1 label to *core* statements and -1 label for *non-core* statements. The evaluator assigns a value of 0 is to any statements that they are not sure about regarding their relevance to the core functionality. To break ties between any disagreements between the two evaluators, we define and apply a rule. For conflicts involving *gray* statements, the other *core* or *non-core* label applies. For conflicts between *core* and *non-core* judgements, we update the label to 0.

C. Interpreting CodeBERT's Predictions Workflow

Figure 2 gives an overview of the process of generating explanations of CodeBERT's predictions from the Python and Java code pairs. The *CloneSHAP* interpreter module shown in the Figure 2, processes one code pair at a time. A code pair has two methods shown as Code$_{m_1}$ and Code$_{m_2}$. Each method has a method header h and a set of statements $s_1, ..., s_n$. Only one of the methods of a code pair, whose SHAP values need to be generated, gets mutated at a time, while the code for the other method remains in its original form. For getting statement permutations for a method, we use the *KernelExplainer* [25] from the Python SHAP library. Kernel SHAP is a method that uses a special weighted linear regression to compute the importance of each feature. The grey blocks in the Code$_{m_i}$ permutations matrix indicate the statements not included in some permutation. We then convert each statement permutation to the embedding vector space of the input method consisting of statement token IDs. The explainer then gets predictions from CodeBERT for each permutation vector, computes the weight for each permutation with the SHAP kernel, fits the weighted linear model, and returns SHAP values. SHAP values for method m_1 ($SHAP_{m_1}$ in Figure 2) are generated when method m_1 is being mutated. Similarly SHAP values for method m_2 are generated during the mutation of m_2. Since clone detection is a binary classification task, the SHAP KernelExplainer outputs two sets of SHAP values in a list. We choose the set of SHAP values from the index corresponding to the model prediction. These SHAP values are between -1 and +1. A positive SHAP value means the statement has a positive impact on a predicted label. A negative SHAP value means negative impact on a predicted label.

Using the SHAP values (e.g. $SHAP_{m_1}$) and human-assigned statement labels (e.g. $HumanLabel_{m_1}$), we can perform a human-machine correlation analysis to evaluate the CodeBERT model's trustworthiness. Furthermore, we can perform misprediction analysis by analyzing positive SHAP values (e.g. 0.07 in $SHAP_{m_1}$ highlighted in figure) for mispredicted code pairs to identify statements (e.g. s_4 and s_7 in $Code_{m_1}$ highlighted in figure) causing the mispredictions.

IV. Evaluation Methodology

Our evaluation aims to understand and explain CodeBERT's clone detection predictions using the SHAP machine-interpretability technique with respect to human's labels. In relation to the key research questions introduced in Section I, the objectives of our evaluation are to answer the following specific questions:

- **RQ1:** What is the prediction accuracy of CodeBERT for detecting semantic clone and non-clone code pairs?

- **RQ2:** What is the impact of various code statements on CodeBERT's predictions?
- **RQ3:** Does CodeBERT's interpretation of the impact of a statement on the prediction align with the human interpretation?
- **RQ4:** How can we identify causes of CodeBERT's mispredictions using SHAP values?

For RQ1, our goal is to evaluate CodeBERT's semantic clone predictions for Java and Python clone pairs against our revised and reliable clone labels. This will establish the current state of performance of CodeBERT for detecting semantic code clones.

For RQ2, our aim is to explore whether we can identify some parts of code, specifically method headers and statements, that contribute towards a true prediction, and those that contribute towards a false prediction. To achieve this, we group the SHAP values by statement types. Then, we analyze the distribution of SHAP values across these statement types and identify the top statement types that influence the model's prediction. This allows us to interpret the behavior of CodeBERT in terms of its attention to particular types of statements.

The purpose of RQ3 is to establish trustworthiness of CodeBERT for code clone detection by evaluating whether the model analyzes code similar to a human. We do this by checking the closeness of machine interpretation to human interpretation. To do this, humans label essential or core statements that contribute to the core functionality of a method. The SHAP technique allows the identification of the key statements that contribute to the predictions. We compare the human labels against SHAP values. Our intuition is that if the machine-identified *core* statements correspond to the human-labeled *core* statements then we can say that the model is looking at core features similar to human intuition. In case there is little or no correspondence, then it is a cause for concern over the correctness of the model.

The goal of RQ4 is to isolate the causes of mispredictions. By comparing the model's predictions against the human labels, we identify the mispredicted clone pairs. For the mispredicted clone pairs, we analyze the statements using SHAP values as a guide to identify the statements contributing to the mispredictions.

A. Evaluation Metrics

We calculate the precision, recall and accuracy of CodeBERT for semantic clone detection for Java and Python clone pairs using the following formulas for precision (Eq. 1), recall (Eq. 2), and balanced accuracy (Eq. 3):

$$precision = \frac{TP}{TP + FP} \tag{1}$$

$$recall = \frac{TP}{TP + FN} \tag{2}$$

$$balanced\_accuracy = \frac{TPR + TNR}{2} \tag{3}$$

Because of the imbalanced number of clone and non-clone code pairs, we apply the balanced accuracy formula to get the accuracy of classification. A True Positive (TP) classification occurs when both the human clone label and model prediction are a *clone*, a True Negative (TN) occurs when both the human clone label and model prediction are a *non-clone*, False Positive (FP) occurs when the model predicts a *clone* but the human label is a *non-clone*, and False Negative (FN) occurs when the model predicts a *non-clone* but the human label is a *clone*. TPR is the True Positive Rate or recall and TNR is the True Negative Rate or specificity (Eq. 4),

$$TNR = \frac{TN}{TN + FP} \tag{4}$$

We calculate correlations between human-assigned importance levels of statements and machine-interpreted SHAP magnitudes of statements. We calculate the human-machine correlation *C* using Pearson's correlation as shown in the equation 5:

$$C = corr(H_x, SHAP_x) \tag{5}$$

where $x = \{m1, m2\}$ represents the method of the clone pair whose statement label distributions are evaluated for correlation. H_x is the set of statement labels assigned by humans to a method x and $SHAP_x$ is the set of magnitudes of SHAP values generated for statements within a method of a code pair. For example, $H_x = (-1, 1, 1, 1, 1, 0, 1, 1)$ and $SHAP_x = (0.06, 0.05, 0.05, 0.07, 0.04, 0.06, 0.07, 0.05)$.

We calculate the percentage of positive correlations from the total number of clone pairs *S*.

$$PositiveCorrPercent = \frac{[C]}{\mid S \mid} * 100 \tag{6}$$

where $[C]$ is the count of positive correlations for sample S.

V. Evaluation Results

A. CodeBERT's performance analysis

We analyze the CodeBERT model's performance on our sample of 115 Java and 120 Python code pairs from the SemanticCloneBench using our revised and reliable clone labels as ground truth. All code pairs in the SemanticCloneBench dataset are supposed to be true clone pairs. However, we identified 49 Java and 42 Python code pairs as non-clones. CodeBERT has a 67.3% and 62.5% precision of clone detection on Python and Java code pairs respectively. Furthermore, CodeBERT has a 92.3% and 68.18% recall of clone detection on Python and Java code pairs respectively. We find that the balanced accuracy is 56.5% on Java and 54.45% on Python clone pairs.

There are a total of 49 clone pairs in our filtered Java dataset of 115 clone pairs, which are mislabeled in the benchmark (where human label is *F* in Table I). We note that for Java clone pairs, CodeBERT predicts both clones and non-clones correctly 58.26% of the time. Of the 49 mislabeled clone pairs in our sample, 22 are correctly identified by CodeBERT

TABLE I: CodeBERT's predictions for 115 Java and 120 Python clone pairs with respect to human labels

Description	CodeBERT's Prediction	Human Label	Java	Python
True Negative	F	F	22	7
False Negative	F	T	21	6
False Positive	T	F	27	35
True Positive	T	T	45	72

with respect to human labels. Thus CodeBERT can detect the mislabeled clones 44.8% of the time.

We note that for Python clone pairs, CodeBERT predicts clone and non-clone pairs correctly 65.83% of the time. We see a large number of false positive predictions for Python (35) than false negative predictions (6) which means that CodeBERT is biased towards finding similarities in Python clone pairs. For the 42 mislabeled clone pairs in our sample of 120 Python clone pairs which are mislabeled, only seven are correctly identified by CodeBERT. The low accuracy may be attributed to mislabelled clones in the data on which CodeBERT is trained.

Answer to RQ1: We find that CodeBERT has a semantic clone detection prediction accuracy of 56.5% on Java and 54.45% on Python clone pairs.

B. Code Parts Influencing CodeBERT Predictions

We investigate the types of code statements which are influential in causing correct or incorrect predictions. Figure 3 shows the violin plots of distributions of SHAP values over various statement types from both Java and Python code pairs. We observe that for Java clone pairs for which the model prediction is correct, all statement types have a stronger positive impact on the predictions than a negative impact. The highest positive impact on the prediction is seen from the *variable declaration statement*. On the other hand, some of the variable declaration statements may also have a negative impact on the prediction. *Expression statements* have the strongest negative SHAP value which means that such expression statements in true clones may be contributing to functionality different from the core common functionality while in false clones, such statements may be contributing to similar functionality. Based on our comparison of SHAP value averages over statement types for correct predictions, the highest average SHAP value comes from *method headers*. Thus, we may infer that method headers contribute the most on average towards the prediction of a Java clone pair. Indeed, humans also attribute a part of their understanding of a method's functionality from the method headers.

We observe that for Java code pairs for which the model prediction is incorrect, the highest positive impact on the prediction is seen from the *expression statement*, implying that some expression statements contribute significantly to some incorrect predictions. The highest negative impact on the prediction is seen from the *variable declaration statement* (*vardec*). The highest average SHAP value for incorrect predictions comes from *for statements*. Thus, we may infer that *for statements* contribute the most on average towards the misprediction of a Java code pair. Overall, *method headers* and *for statements* have the highest average impact on CodeBERT's correct predictions and incorrect predictions (+0.0638 and +0.09) respectively for Java code pairs.

For Python code pairs for which the model prediction is correct, the highest positive impact on the prediction is seen from the *method header*. On the other hand, the highest negative impact on the prediction is seen from the variable declaration statements. The highest average SHAP value for correct predictions comes from *return statements*. Thus, we may infer that return methods contribute the most on average towards the prediction of a Python code pair. For Python code pairs for which the model prediction is incorrect, the highest positive and negative impacts on the prediction is seen from the *variable declaration statements*. The highest average SHAP value for mispredictions comes from *return statements*. Overall, *return statements* have the highest average impact on CodeBERT's correct and incorrect predictions (+0.070 and +0.074 respectively) for Python code pairs.

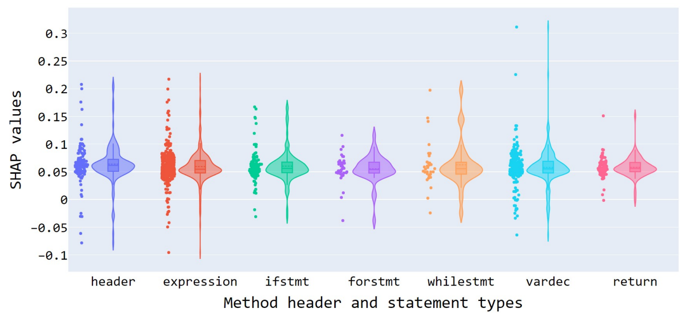
Answer to RQ2:
Method headers and *for statements* have the highest average impact on CodeBERT's predictions for Java clone pairs, whereas for Python the impact is from *return statements*.

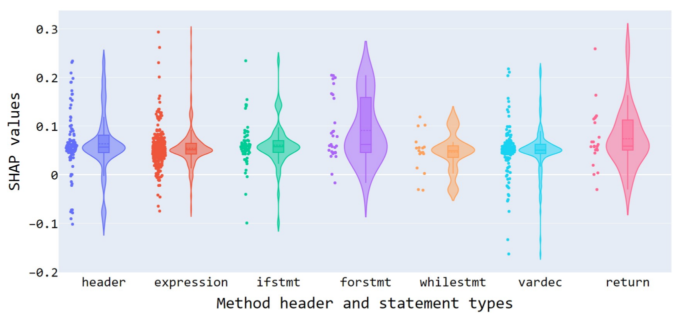
C. Human-Machine Interpretation Correlation Analysis

We analyze the SHAP values for *core*, *non-core* and *gray* statements labeled by humans to see whether *core* statements have more impact on CodeBERT's predictions than *gray* statements, which in turn, have a higher impact on predictions than *non-core* statements. From Figure 4 we see that there is no such trend for Java code. For Java clone pairs, all core, non-core, and gray statement types have similar SHAP value distributions indicating similar impact on predictions. On the other hand, we note that for Python code, the magnitude of impact of *core* statements is noticeably greater than for *non-core* statements. We also note that the gray statements possess more positive SHAP values than negative values. Furthermore, the non-core statements have lower positive impact than gray statements. This indicates that for Python code, the model's predictions are more affected by core statements similar to our human intuition.

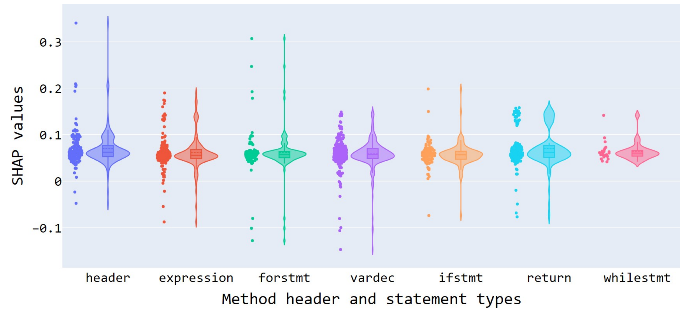
For evaluating the human-machine correlation, we compare the magnitudes of SHAP values of statements with the human evaluators' assigned labels using Pearson's correlation analysis. Table II shows the percentage of positive correlations between human-assigned statement labels and SHAP magnitudes for a sample size $|S|$ of Java and Python clone pairs.

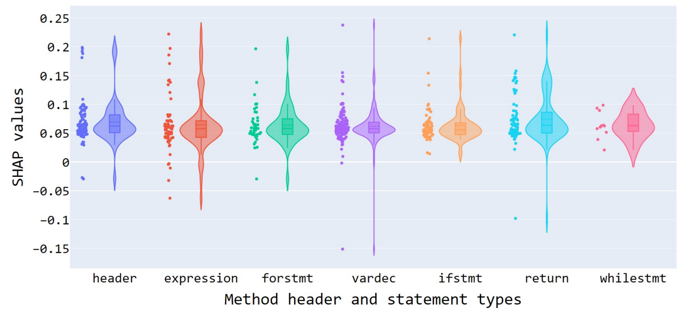
Table II shows the human and machine interpretation correlations over a sample S of clone pairs for Java and Python.





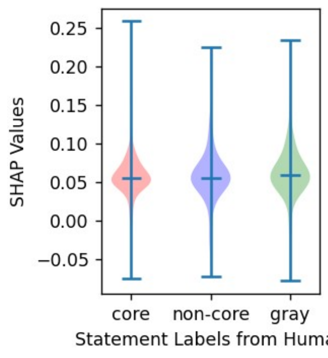
(a) Java code statements contribution towards correct (left) and incorrect (right) model predictions



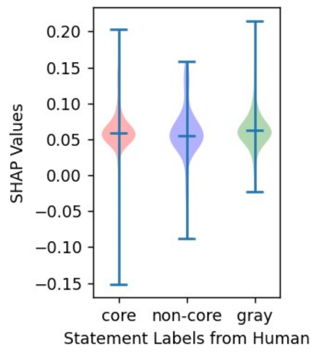


(b) Python code statements contribution towards correct (left) and incorrect (right) model predictions

Fig. 3: Visualizing Java and Python code statements contribution towards correct and incorrect model predictions



(a) Java code



(b) Python code

Fig. 4: Visualizing human and model intuition correspondence from a distribution of SHAP values for different statement labels

Each row in the table shows the percentage of positive correlations between the human and model distributions for the first method (m_1) or second method (m_2) a code pair. As we mentioned earlier that we obtain the SHAP values of each method in a code pair independently, by keeping one method intact while mutating the other. Each row in the table presents human-machine correlations for code pairs of a certain code (Java/Python) and prediction outcome (correct/incorrect).

For Java clone pairs, we observe that for 53.22% of the clone pairs, the SHAP-based machine-interpretation for m_1 has a positive correlation with the human-interpretation (shown in Table II) in case of correct predictions. Also, for 48.33% of the code pairs, the SHAP-based machine-interpretation for m_2 has a positive correlation with the human-interpretation in case of correct predictions. However, the meaningful correlations where the correlation coefficient is higher than 0.5 are rare with 12.9% for m_1 intuitions and 11.66% for m_2 intuitions. We observe that the correlation is weaker for incorrect predictions as expected at only 4.34% for m_1 intuitions and 2.32% for m_2 intuitions. We can infer that the CodeBERT model perceives the Java statements in a way different from a human.

For Python code pairs, we observe that for 43.08% of the clone pairs, the SHAP-based machine-interpretation for m_1 has a positive correlation with the human-interpretation (shown in Table II) in case of correct predictions. Also, for 51.56% of the code pairs, the SHAP-based machine-interpretation for m_2 has a positive correlation with the human-interpretation in case of correct predictions. However, the meaningful correlations where the correlation coefficient is higher than 0.5 are rare with 10.77% for m_1 intuitions and 6.25% for m_2 intuitions. We observe that the correlation is weak for incorrect predictions at only 8.1% for m_1 intuitions and 8.33% for m_2 intuitions. We can infer that the CodeBERT model perceives the contribution and impact of Python statements in a way different to a human.

Answer to RQ3: Machine-interpretation of statements correlates with human-intuition only as much as 53.2% of the time for Java and 58.33% of the time for Python clone pairs.

TABLE II: Human (H) and machine (M) interpretation correlations (Cor.) for Java and Python clone pairs grouped by correct (✓) and incorrect (X) predictions

Code	H	M	Prediction	$\%PositiveCor.$	% Cor. $\geq$ 0.5	$\|S\|$
Java	H_{m1}	$SHAP_{m1}$	✓	53.22	12.9	62
Java	H_{m1}	$SHAP_{m1}$	X	45.65	4.34	46
Java	H_{m2}	$SHAP_{m2}$	✓	48.33	11.66	60
Java	H_{m2}	$SHAP_{m2}$	X	50.00	2.32	43
Python	H_{m1}	$SHAP_{m1}$	✓	43.08	10.77	65
Python	H_{m1}	$SHAP_{m1}$	X	56.76	8.1	37
Python	H_{m2}	$SHAP_{m2}$	✓	51.56	6.25	64
Python	H_{m2}	$SHAP_{m2}$	X	58.33	8.33	36

D. *Misprediction Analysis*

In Figure 5, we present a Java clone pair that is mispredicted by CodeBERT when in fact it is a true semantic clone pair. The methods have some similarities in terms of prompting the user for input and validating it, but they also have some differences in their implementation. The first function reads input using *input.findInLine(".")* to check if a dot character is found, and then retrieves the first character of the input as the choice. It uses a *while* loop to validate the choice and prompt the user for a valid input until a valid choice of 'x' or 'o' is entered. The second function also reads input using *input.findInLine(".")* to check if a dot character is found. If a dot character is found, it retrieves the first character of the input as the choice. However, it uses an *if* condition to conditionally enter a *while* loop for validation and prompting the user for a valid input. This loop is only entered if the dot character is found and the choice is not 'x' or 'o'.

The SHAP values of the statements for the first method are $SHAP_{m_1}$ = (0.06, 0.05, 0.05, 0.07, 0.04, 0.06, 0.07, 0.05) and for the second method they are $SHAP_{m_2}$ = (0.10, -0.16, -0.07, 0.03, 0.11, 0.10, 0.08, 0.23, 0.06). From the values in $SHAP_{m_1}$, we note that the fourth and seventh statements have the highest positive SHAP values, indicating that these statements on Line 4 and Line 7 of Listing 3 have the highest impact on predicting it as a non-clone. It is interesting to note that the code on Lines 4 and 6 of the first method is different from the second method. From the values in $SHAP_{m_2}$, we note that the eighth statement has the highest positive SHAP value, indicating that this statement on Line 8 of Listing 4 has the highest impact in predicting the clone pair as a non-clone. On the other hand, the negative SHAP values corresponding to the second and third statements have the highest impact on predicting it as a clone. It is interesting to note that the code on Lines 2 and 3 of the second method is similar to code in the first method.

In Figure 6, we present a Python clone pair that is mispredicted by CodeBERT when in fact it is a true semantic clone pair. These two methods are similar in their purpose, which is to retrieve a value from a nested structure based on a given path. The first method iterates over each level in the path and uses the *get( )* method to retrieve the corresponding value from the matrix. If the value is not found at any level, it breaks out of the loop and returns *None*. The second method uses the *reduce( )* function and *operator.getitem()* to traverse

```
1  char wf () {
2      Scanner input = new Scanner (System.in);
3      System.out.println ("What is your choice? (x/o)");
4      char choice = input.findInLine (".").charAt (0);
5      while (choice != 'x' && choice != 'o') {
6          System.out.println ("You must enter x or o!");
7          choice = input.next ().charAt (0);
8      }
9      return choice;
10 }
```

Listing 3: Clone29 - A

```
1  char wf () {
2      Scanner input = new Scanner (System.in);
3      System.out.println ("What is your choice? (x/o)");
4      if (input.findInLine (".") != null) {
5          choice = input.findInLine (".").charAt (0);
6          while (choice != 'x' && choice != 'o') {
7              System.out.println ("You must enter x or o!");
8              choice = input.findInLine (".").charAt (0);
9          }
10     }
11     return choice;
12 }
```

Listing 4: Clone29 - B

Fig. 5: Java clone pair mispredicted as false by CodeBERT. Statements highlighted have high SHAP values indicating the cause of misprediction.

the nested structure and retrieve the value at the specified path. If a *KeyError* is raised during the traversal, indicating that a level in the path is not found, it returns *None*. Both functions provide a way to access nested values based on a given path, but they differ in their implementation and thus are semantic clones. The SHAP values of the statements for the first method are $SHAP_{m_1}$ = (0.03, -0.03, 0.06, 0.05, -0.02, 0.02) and for the second method they are $SHAP_{m_2}$ = (-0.03, 0.18, -0.09, -0.13, 0.22). From the values in $SHAP_{m_1}$ and $SHAP_{m_2}$, we note that the third and fifth statements have the highest positive SHAP values for m_1 and m_2 respectively, indicating that these statements in Listing 5 and Listing 6 have the highest impact on predicting it as a non-clone. It is interesting to note the statements with negative SHAP values like the header in $m2$ and other statements that are pushing the prediction towards a clone prediction. In this way using SHAP values, we can interpret the causes of misprediction for a particular clone pair.

Answer to RQ4: Statements having the highest positive SHAP values can be identified as the major contributors towards a misprediction.

VI. Discussion

From the answer to RQ1, we note that the clone detection performance of CodeBERT on semantic clones that we label is lower from what is previously reported. Thus we note that CodeBERT's clone detection performance does not generalize well to a new dataset.

From the answer to RQ2, we found that *method headers* and *for statements* have the highest average impact on

```
def get_target_path(pth, mtx) :
   for level in pth :
      mtx = mtx.get(level, None)
      if mtx is None :
         break
   return mtx
```

Listing 5: Clone314.py - A

```
def get_target_path(path, matrix) :
   try :
      return reduce(operator.getitem, path, matrix)
   except KeyError :
      return None
```

Listing 6: Clone314.py - B

Fig. 6: Python clone pair mispredicted as false by CodeBERT. Statement highlighted have high SHAP values contributing towards the misprediction.

CodeBERT's predictions for Java clone pairs. This means the CodeBERT model trained itself to pay more attention to *method headers* and *for statements* while making decisions on code similarity. For Python clone pairs, CodeBERT seems to have trained itself to decide the clone label by paying attention to *return statements*. Thus, we discover some intrinsic training patterns for the CodeBERT model.

From the answer to RQ3, we note that the impact magnitudes (SHAP values) of statements on CodeBERT's predictions do not correspond to the importance levels that humans assign to statements. Thus, we infer that CodeBERT's intuition for predicting clones does not align with human intuition.

We demonstrated how SHAP values may help us identify the statements that may be the reason for the mispredictions. While we only discuss cases where CodeBERT mispredicts a true clone pair, it would be interesting to analyze the cases where CodeBERT mispredicts a false clone pair as well. Due to limited space, we leave this discussion for future work.

VII. Threats to Validity

A. Construct Validity

Our study acknowledges a potential limitation concerning the labeling of clones and code statements. We followed a labeling guideline that we agreed upon within our research team. However, it is important to note that the definition of semantic clones lacks strict consensus within the code clone community. Therefore, what we identify as a clone based on our criteria may not align with the definition or identification of clones used by other researchers or practitioners. Furthermore, the importance levels for statements are subjective to the intuition of the evaluators and may be different for other Java and Python experts.

SHAP has its own limitations and using the SHAP technique is a threat to validity of our interpretation of CodeBERT's behavior. This threat comes from how SHAP values are approximated. KernelSHAP works by permuting feature values and making predictions on those permutations. The problem is, when permuting features (statements in our case), we assume they are independent. However with code, we have dependencies across statements. Furthermore, the SHAP value method suffers from inclusion of unrealistic data instances while generating and evaluating permutations of code. These two issues pose a threat to the validity of the conclusions which we make regarding the impact of statements on predictions. The use of better interpretation techniques for clone detection models is needed for future work.

Our correlation analysis measures for correspondence between human and model intuition focus only on positive correlations only and may not fully reflect all kinds of correspondence.

B. External Validity

The filtering of clone pairs with methods having more than ten lines of code is a threat to the validity of our conclusions for larger methods. Whether our results on the current smaller code examples can generalize to larger code is future work.

C. Internal Validity

While we strive to include major statement types that are likely to contain core functionality, the analysis results from including other statement types may be different from our current findings for Java. While we consider the impact of expression statements as a whole, there may be different impacts for sub-types of expressions.

VIII. Related Work

Interpretability techniques for machine learning models can be classified into two categories, namely local and global interpretability. Local explainability techniques aim to provide justification for predictions on a sample input [18, 26–29]. On the other hand, global explainability techniques aim to explain the behavior of the whole model [18, 30–32]. In our work, we apply local interpretatbility on individual clone pair samples and then perform a cumulative analysis by aggregating the local interpretability results to achieve global interpretability.

There are different code mutation or simplification techniques [33–36] used for understanding code models for different tasks; however, our interpretation focuses on semantic clone prediction behaviors with respect to human understanding which is the first of its kind. Apart from SHAP, another related perturbation-based explanation mechanism is LIME [26]. LIME fits a local interpretable model to perturbed input instances to explain an input's prediction. SHAP is the better approach; having a strong theoretical foundation in cooperative game theory, which provides clear rationale for feature importance calculations.

IX. Conclusion

In this paper, we present a novel method using SHAP values to explain CodeBERT's predictions for classifying semantic code clones. SHAP values help in identifying the key code statements in a clone pair that influence the prediction. Such explanations of the predictions helps us to identify the statements that the model picks on to determine semantic similarity. Furthermore, explanations of mispredictions help to identify characteristics of data samples for fine-tuning and improving CodeBERT's performance. Our analysis reveals that machine-interpretation of core statements is weakly correlated to human-intuition indicating a gap between human and model interpretation. Other researchers can use our labeled dataset of semantic clones and our interpretation method to evaluate other clone detection models and interpret their models' clone detection behavior and assess their model's reliability. To support the replication of our experiment and results, we have uploaded the code for our interpretation techniques and the results of our experiments online on GitHub [19].

References

[1] Chanchal Kumar Roy and James R Cordy. A Survey on Software Clone Detection Research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[2] Abdullah Sheneamer and Jugal Kalita. Semantic clone detection using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1024–1028. IEEE, 2016.

[3] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[4] Huihui Wei and Ming Li. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *IJCAI*, pages 3034–3040, 2017.

[5] Gang Zhao and Jeff Huang. DeepSim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 141–151, 2018.

[6] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365, 2018.

[7] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 542–553. IEEE, 2018.

[8] Abdullah Sheneamer. CCDLC Detection Framework-Combining Clustering with Deep Learning Classification for Semantic Clones. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 701–706. IEEE, 2018.

[9] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 70–80. IEEE, 2019.

[10] Yuan Yuan, Weiqiang Kong, Gang Hou, Yan Hu, Masahiko Watanabe, and Akira Fukuda. From local to global semantic clone detection. In *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, pages 13–24. IEEE, 2020.

[11] Bingzhuo Li, Chunyang Ye, Shouyang Guan, and Hui Zhou. Semantic Code Clone Detection Via Event Embedding Tree and GAT Network. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 382–393. IEEE, 2020.

[12] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 516–527, 2020.

[13] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv preprint arXiv:2009.08366*, 2020.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155*, 2020.

[15] Tim Sonnekalb, Bernd Gruner, Clemens-Alexander Brust, and Patrick Mäder. Generalizability of code clone detection on codebert. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[16] Saad Arshad, Shamsa Abid, and Shafay Shamail. Codebert for code clone detection: A replication study. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 39–45. IEEE, 2022.

[17] Farouq Al-Omari, Chanchal K Roy, and Tonghao Chen. Semanticclonebench: A semantic code clone benchmark using crowd-source knowledge. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 57–63. IEEE, 2020.

[18] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

[19] CloneSHAP Interpreter Code, Data and Results. https://github.com/Cxm211/CloneSHAPInterpreter, July 2023.

[20] Stackoverflow. https://stackoverflow.com/, February 2022.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[22] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692*, 2019.

[23] Telmo Pires, Eva Schlinger, and Dan Garrette. How multilingual is Multilingual BERT? *arXiv preprint arXiv:1906.01502*, 2019.

[24] BigCloneBench-Dataset. https://github.com/microsoft/CodeBERT/blob/master/GraphCodeBERT/clonedetection/dataset.zip, February 2021.

[25] SHAP KernelExplainer. https://shap.readthedocs.io/en/latest/example_notebooks/tabular_examples/model_agnostic/Simple%20Kernel%20SHAP.html?highlight=kernelexplainer#Simple-Kernel-SHAP, December 2022.

[26] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1135–1144, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. Counterfactual explanations for models of code. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, page 125–134, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Ehud Kalai and Dov Samet. On weighted shapley values. *International journal of game theory*, 16(3):205–222, 1987.

[29] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[30] Jürgen Cito, Isil Dillig, Seohyun Kim, Vijayaraghavan Murali, and Satish Chandra. Explaining mispredictions of machine learning models using rule induction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 716–727, 2021.

[31] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[32] Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Jure Leskovec. Interpretable & explorable approximations of black box models. *arXiv preprint arXiv:1707.01154*, 2017.

[33] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. Understanding neural code intelligence through program simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 441–452, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Md Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. Syntax-guided program reduction for understanding neural code intelligence models. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, jun 2022.

[35] Md Rafiqul Islam Rabin and Mohammad Amin Alipour. Programtransformer: A tool for generating semantically equivalent transformed programs. *Software Impacts*, 14:100429, 2022.

[36] Md Rafiqul Islam Rabin and Mohammad Amin Alipour. Featureextractor: A tool for extracting key input features of code intelligence models. *Software Impacts*, 14:100432, 2022.