Code Pair Labeling Guidelines

Introduction:

The purpose of this document is to provide detailed guidelines for human annotators tasked with labeling code pairs. Code pair labeling can be performed with the VSCode IDE using our CodeCloneAnnotation Tool available on GitHub[1]. A code pair consists of two methods, and the labeling process involves interpreting the functionality of each method within the context of the pair and performing:

- Clone labeling
- Code Labeling

Clone Label Values

A *clone label* refers to the label assigned to a code pair after evaluation by a human. A clone label can take on one of two values: `clone' or `non-clone', representing whether a code pair contains methods with semantically similar functionality (`clone') or different functionality (`non-clone').

Code Label Values

A *code label* refers to the label assigned by a human evaluator to some selected portion of code in a code pair. A code label can take on one of four values; where a code label value represents either a core similarity, a non-core similarity, a core difference, or a non-core difference.

Portions of code that can be assigned code labels may be as small as a single token and may span multiple lines. A code label must be assigned to a contiguous portion of code selected within a method body of a code pair, ensuring that the selection does not span across two methods.

Following is a description of each code label type:

- CoreSim
    - This type of code chunk implements the core functionality of the method and a corresponding code chunk with the same core functionality can also be found in the other method of the code pair. For example, if a method contains code to sort an array of integers, then the statements directly manipulating the array and performing the sort would be the core statements.
- NonCoreSim
    - This type of code chunk implements some non-core or supplementary functionality of the method and a corresponding code chunk with the same core functionality can also be found in the other method of the code pair. For example, if a method contains code to sort an array of integers and has some exception handling code, logging code like print statements or some variable declarations or object initializations would be considered as non-core.
- CoreDiff
    - This type of code chunk implements the core functionality of the method and a corresponding code chunk with the same core functionality can also be found in the other method of the code pair. For example, if a method contains code to sort an array

[1] https://github.com/shamsa-abid/Code-Clone-Causal-Interpretation/tree/main/CodeAndCloneAnnotationTool

of integers, then the statements directly manipulating the array and performing the sort would be the core statements.

- NonCoreDiff
    - This type of code chunk implements some non-core or supplementary functionality of the method and can only be found in one method of the code pair. For example, if a method contains code to sort an array of integers and has some exception handling code, logging code like print statements or some variable declarations or object initializations that are not found in the other method, these would be considered as non-core differences.

Methods of Interpretation:

1. **Reading Code**: Annotators should carefully read the code, including the name of each method, the input parameters, and the statements comprising the method body. Pay close attention to the logic flow and any conditional statements within the method.
2. **Performing a Dry Run**: Conduct a dry run of the code to determine its behavior and functionality. Trace the execution path and identify any key operations or transformations performed by the code.
3. **Referring to Stack Overflow Posts**: Optionally refer to any Stack Overflow posts associated with the code pair. Clarify any confusion regarding the functionality of the method or its intended behavior by reviewing discussions and explanations provided by the community.
4. **Executing the Code**: In cases where the code is complex or obscure, consider executing the code to verify its output. This step can help validate the behavior of the code and ensure a more accurate interpretation of its functionality.

Labeling Process:

1. **Broad/Bird's Eye Analysis:**
    a. Use the methods of interpretation to evaluate each of the two methods (m1 and m2) within a code pair.
    b. After the interpretation of each method's functionality, compare both functionalities, and decide whether the functionalities are similar or different.
    c. If both m1 and m2 are functionally similar and self-contained (do not call an external function containing the actual functionality), label the code pair as a *clone*.
    d. If both m1 and m2 are similar and are not self-contained, label the code pair as '*clone-ext'*.
    e. If both m1 and m2 are functionally dissimilar because their interpretations don't match, and are self-contained, label the code pair as *'non-clone'*.
    f. If both m1 and m2 are functionally dissimilar because their interpretations don't match, and are not self-contained, label the code pair as '*non-clone-ext'*.
2. **Fine-grained Analysis:**
    a. Use the methods of interpretation to evaluate each of the two methods (m1 and m2) within a code pair.
    b. After interpretating the functionality of each of the two methods of a code pair, label the code segments at a fine-grained level.

i. First identify and label any corresponding core similarities and non-core similarities across the two methods.
ii. Then for any remaining unlabeled code, identify and label any core differences and non-core differences across the two methods.

c. Determine whether the methods in the code pair contain corresponding similarities, or differences, or a combination of both. Use the Table below listing various combinations of the presence and absence of core similarities and differences to assign the clone label.

Case ID	$coresim_{m1}$	$corediff_{m1}$	$coresim_{m2}$	$corediff_{m2}$	Clone Label
1	✓	x	✓	x	clone/clone-ext
2	✓	✓	✓	✓	Intuitive reasoning
3	✓	✓	✓	x	Intuitive reasoning
4	✓	x	✓	✓	Intuitive reasoning
5	x	✓	x	✓	Non-clone/Not-clone-ext

d. For case IDs 1 and 5, deciding the clone label is straightforward. Assign a *clone* label if the functionalities are self-contained and assign a *clone-ext* label if the functionalities are not self-contained for either of the methods.

e. For case IDs 2,3 and 5, there is a combination of core similarities and differences across the two methods of a code pair. Use intuitive reasoning and assign a 'clone' label only if the portions of code containing similarities are greater than the portions containing dissimilarities. Assign a 'clone-ext' label if the functionalities are not self-contained for either of the methods and only if the portions of code containing similarities are greater than the portions containing dissimilarities. Otherwise, if the portions of code containing similarities are lesser than the portions containing dissimilarities, assign a 'non-clone' label if the functionalities of the two methods are self-contained or 'non-clone-ext' label if the functionalities are not self-contained.

3. **Documenting Rationale**: Document the rationale behind the assigned labels, including key observations, reasoning, and any additional context that influenced the labeling decision. This documentation will serve as valuable feedback for future revisions and quality assurance checks.

Conclusion:

These guidelines aim to facilitate a systematic and rigorous approach to labeling code pairs, ensuring the creation of high-quality datasets for software engineering tasks. By following these guidelines,

annotators can contribute to the development of reliable and comprehensive datasets that support various research and practical applications in the field.

Note: *These guidelines are subject to updates and revisions based on feedback and evolving best practices in code labeling methodologies.*