**1. Methodology**

**1.1 Dataset Description**

The dataset was referred from [research paper]. The dimension of the dataset is 512x177, and it contains commit hashes from the Linux open source Github Repository. The list of important features we will be using for this research paper and their description are given in Table 1.

|  |  |
| --- | --- |
| Feature | Description |
| Description | Contains the commit hash, File Path and the function name, separated by an underscore |
| Commit | Contains the commit hash from GitHub |
| FilePath\_y | File path of the file changes on GitHub |
| function\_name\_cfg | The name of the function which is to be evaluated |
| Binary\_label | It only has 0s and 1s telling us whether the function in the file has been changed or not |

**1.2 Preprocessing**

The dataset was preprocessed and unwanted features were removed. Rows with no values were also removed. During the preprocessing phase, I had also added a column which acts as a primary key while saving the .cpp files.

**1.3 Extract and Save .CPP files**

I have used GitHub API’s to extract information using the commit hash.

I have made a code that uses GitHub API to extracts commit information from a GitHub repository and saves the contents of `.cpp` files to a local folder. The detailed explaination is as follows

Extracting Commit Information

1. Initialize GitHub API Requests - The code uses the GitHub API to fetch details about specific commits, trees, and file contents. It requires the repository owner, repository name, commit hash, file path, and a GitHub access token for authentication.
2. Fetching Commit Details - The `get\_commit\_details` method constructs a URL using the repository owner, name, and commit hash. It sends a GET request to this URL, including the access token in the headers which prevents rate limit which is set for the GitHub API if the access token is not used. If the request is successful (status code 200), it returns the commit details as a JSON object. Otherwise, it raises an exception.
3. Fetching the Tree - The `get\_tree` method constructs a URL to fetch the tree structure of the repository using the tree SHA from the commit details. It sends a GET request with the access token. If successful, it returns the tree structure as a JSON object. Otherwise, it raises an exception.
4. Finding a File in the Tree - The `find\_file\_in\_tree` method iterates through the items in the tree to locate a specific file by its path. If it finds the file, it returns the file’s SHA; otherwise, it returns `None`.
5. Fetching File Content - The `get\_file\_content` method constructs a URL to fetch the file content using the file SHA. It sends a GET request with the access token and sets the `Accept` header to receive the raw file content. If successful, it returns the file content as text. Otherwise, it raises an exception.
6. Viewing a File in a Commit - The `view\_file\_in\_commit` method combines the above methods to get the commit details, tree SHA, tree structure, and file SHA. It then fetches the file content and returns it. If the file is not found in the tree, it returns `None`.
7. Fetching the Previous Commit - The `get\_previous\_commit` method fetches the parent commit hash from the commit details. If there are no parents, it returns `None` (indicating the initial commit). Otherwise, it returns the SHA of the first parent commit.

Saving .cpp Files

1. Setting Up Folder Structure - The `to\_save\_as\_CPP\_files\_in\_folder` method takes several parameters, including the main folder name, repository details, a primary key column name (used as the file prefix), and a DataFrame containing the commit hash and file path information. It creates the main folder if it doesn’t exist.
2. Iterating Through DataFrame Rows - For each row in the DataFrame, it creates a subfolder named after the primary key column value (`id\_for\_file\_name`).
3. Extracting and Saving Current Commit File Content - For each row, it fetches the current commit file content using the `view\_file\_in\_commit` method. If the file content is found, it writes the content to a `.cpp` file in the corresponding subfolder, using the primary key value as part of the filename.
4. Extracting and Saving Previous Commit File Content - It then fetches the previous commit hash using the `get\_previous\_commit` method. If a previous commit exists, it fetches the previous commit file content. If the file content is found in the previous commit, it writes the content to another `.cpp` file in the same subfolder, again using the primary key value as part of the filename.
5. Handling Errors - If any exceptions occur during the process, they are caught and printed to the console.

This process allows you to programmatically extract and save specific versions of files from a GitHub repository, organizing them in a structured manner based on the primary key from the DataFrame.

**1.4 Create a JSON file to Record Outputs**

The method `create\_json\_file\_format\_1` within the `JsonFunctions` class is designed to generate a JSON file that serves as a template for recording various metrics and attributes related to commit and neutral code. It accepts two parameters: `file\_name`, which determines the name of the JSON file to be created, and `limit`, which specifies the number of version entries to include in the file. For each version, represented as "V\_001", "V\_002", and so on up to the limit, the method initializes a nested dictionary structure with placeholders for "commit\_code" and "neutral\_code". Each of these categories contains fields for storing information about the code's size, complexity, memory management, code complexity, and error handling, all initially set to zero or empty strings. After constructing the data structure, the method writes it to a JSON file with the specified file name, formatted with indentation for readability. This JSON file acts as a structured framework for logging and analyzing code metrics across different versions.

**1.5 Extract Changed Functions and save them in JSON file**

This is a comprehensive workflow implemented to detect and document specific function changes in C++ files between different commits and save these changes in a structured JSON format. The procedure begins by iterating through a DataFrame that includes the paths to the C++ files and the names of the functions intended for extraction. For each row in the DataFrame, the code constructs file paths pointing to the C++ files from the current and previous commits. It utilizes the `read\_cpp\_file` method from the `ParseCPPToExtractFunction` class to read the contents of these files.

When a function name is provided (indicating it is not '0'), the `find\_function\_content` method searches the file content using a regular expression to locate the complete function definition, including its body. This method identifies the function by matching the function name and capturing the surrounding code, ensuring it accurately handles nested braces to find the function's boundaries.

Once the function code from the current commit is extracted, it is added to a JSON file under the "neutral commit" section using the `JsonFunctions.add\_neutral\_commit\_code\_to\_json\_format1` method. This JSON file is structured to organize and log various code metrics and details, ensuring that each function's state is recorded systematically. Similarly, the function code from the previous commit is extracted and added to the JSON file under the "vulnerable commit" section. This dual recording allows for a detailed comparison between the current and previous states of the function, highlighting changes that might indicate potential vulnerabilities or improvements.

Throughout this process, the code handles potential errors robustly. If any C++ files are missing or other errors occur, appropriate error messages are printed to help diagnose and resolve issues promptly. This workflow not only ensures the accurate extraction and documentation of function changes but also facilitates the analysis of code evolution and the identification of potential issues in the development lifecycle. By systematically saving these changes in a JSON file, the code provides a valuable resource for tracking code modifications and assessing their impact on software quality and security.

**1.6 Run Testing on Ollama**

About the parameters in which the tests are done. Describe each parameter. Question formats