BUG Localization

1 Introduction

In the realm of software development, debugging and rectifying software bugs stand out as one of the most time-consuming and energy-draining processes. The journey from bug discovery to resolution typically involves the creation of textual documents known as 'bug reports,' which encapsulate crucial details such as bug description and report time. Once a bug report surfaces, whether initiated by an end user or a test engineer, the responsibility of resolving it falls upon a member of the developer team. At this juncture, the developer assumes the mantle of localizing and rectifying the bug within the codebase.

For developers navigating through the intricacies of a large-scale software project, the arduous task of pinpointing the precise source code files responsible for a bug emerges as a formidable challenge. Often, this involves sifting through a multitude of source code files in pursuit of the elusive culprit(s) behind the software defect. This process, aptly termed bug localization, stands as a significant bottleneck in the bug-fixing endeavor.

Recognizing the inefficiency inherent in manual bug localization, numerous studies have delved into automating this process. The aim is to empower developers by enabling them to focus their efforts on rectifying bugs within localized source code files.

The landscape of bug localization research boasts a variety of approaches. Advanced information retrieval techniques and diverse machine learning methodologies have been harnessed to establish a correlation between bug reports and source code files. This report sheds light on the implementation of two distinct studies aimed at automating the bug localization process.

2 Problem Statement

Software bugs can cause significant disruptions and delays in software development projects. One of the critical challenges in software maintenance is the efficient and accurate localization of bugs. Current bug localization techniques often rely on manual inspection or simple keyword search, which can be time-consuming and inefficient. The aim of this project is to develop an automated bug localization system that can accurately identify the source code files likely to contain the root cause of a reported bug, thereby reducing the time and effort required for bug resolution.

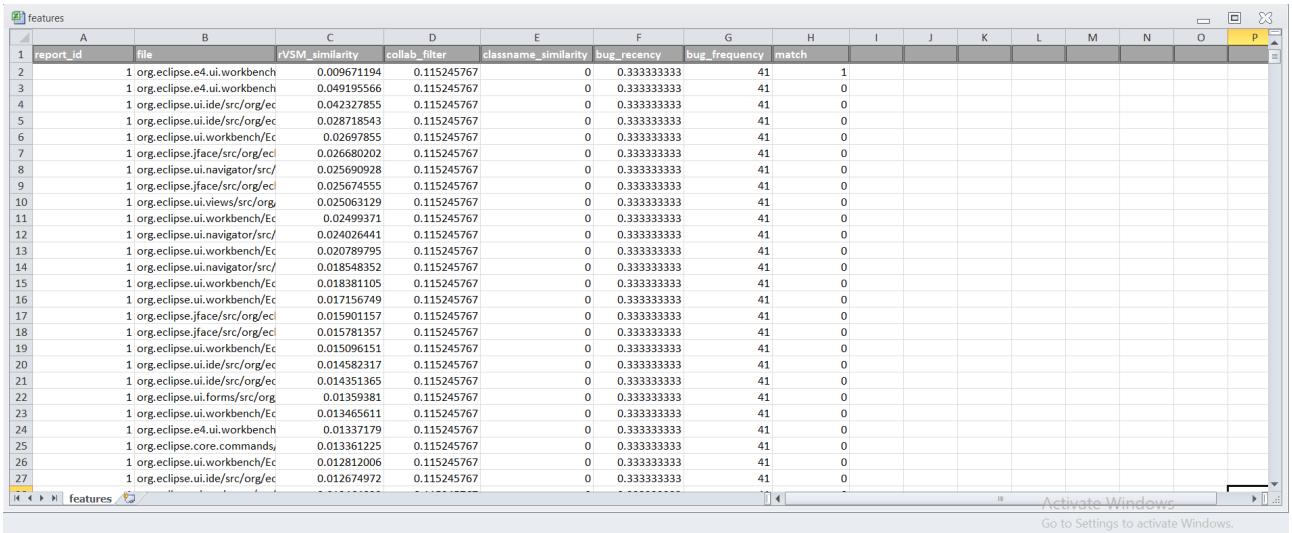
3 Dataset

Despite the abundance of bug localization datasets available online, we decided to utilize a specific dataset for our experiments. This dataset encompasses bug histories from six different projects: AspectJ, Birt, Eclipse Platform UI, JDT, SWT, and Tomcat. For our implementation, we focused solely on the Eclipse Platform UI dataset, which includes 6495 bug reports.

Each bug report within the dataset is characterized by various attributes, including bug ID, summary, description, report time, report timestamp, status, commit, commit time, and file features. A sample excerpt from the dataset is presented in Table 1, showcasing the structured information available for analysis. Access to commit details, commit time, and files is facilitated through the GitHub repository of the respective project. The files section specifies all added, modified, and deleted files associated with each bug report.

To facilitate the extraction of similarity features between bug reports and files, we required access to Java source files of the project in their version preceding the related commit. In total, our dataset encompasses 8735 buggy files and their corresponding bug reports. However, at the onset of our analysis, the features extracted from these files and reports were not available. Hence, we undertook the task of completing all aspects of feature extraction independently.

To enrich the dataset and ensure its comprehensiveness, we also augmented it with additional metadata where applicable, enhancing the depth of analysis and facilitating a more robust evaluation of our implementation's performance.



features

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	report_id	file	rVSM_similarity	collab_filter	classname_similarity	bug_recency	bug_frequency	match								
2	1	org.eclipse.e4.ui.workbench	0.009671194	0.115245767	0	0.333333333	41	1								
3	1	org.eclipse.e4.ui.workbench	0.049195566	0.115245767	0	0.333333333	41	0								
4	1	org.eclipse.ui.ide/src/org/ec	0.042327855	0.115245767	0	0.333333333	41	0								
5	1	org.eclipse.ui.ide/src/org/ec	0.028718543	0.115245767	0	0.333333333	41	0								
6	1	org.eclipse.ui.workbench/Ec	0.02697855	0.115245767	0	0.333333333	41	0								
7	1	org.eclipse.jface/src/org/ecl	0.026680202	0.115245767	0	0.333333333	41	0								
8	1	org.eclipse.ui.navigator/src/	0.025690928	0.115245767	0	0.333333333	41	0								
9	1	org.eclipse.jface/src/org/ecl	0.025674555	0.115245767	0	0.333333333	41	0								
10	1	org.eclipse.ui.views/src/org/	0.025063129	0.115245767	0	0.333333333	41	0								
11	1	org.eclipse.ui.workbench/Ec	0.02499371	0.115245767	0	0.333333333	41	0								
12	1	org.eclipse.ui.navigator/src/	0.024026441	0.115245767	0	0.333333333	41	0								
13	1	org.eclipse.ui.workbench/Ec	0.020789795	0.115245767	0	0.333333333	41	0								
14	1	org.eclipse.ui.navigator/src/	0.018548352	0.115245767	0	0.333333333	41	0								
15	1	org.eclipse.ui.workbench/Ec	0.018381105	0.115245767	0	0.333333333	41	0								
16	1	org.eclipse.ui.workbench/Ec	0.017156749	0.115245767	0	0.333333333	41	0								
17	1	org.eclipse.jface/src/org/ecl	0.015901157	0.115245767	0	0.333333333	41	0								
18	1	org.eclipse.jface/src/org/ecl	0.015781357	0.115245767	0	0.333333333	41	0								
19	1	org.eclipse.ui.workbench/Ec	0.015096151	0.115245767	0	0.333333333	41	0								
20	1	org.eclipse.ui.ide/src/org/ec	0.014582317	0.115245767	0	0.333333333	41	0								
21	1	org.eclipse.ui.ide/src/org/ec	0.014351365	0.115245767	0	0.333333333	41	0								
22	1	org.eclipse.ui.forms/src/org	0.01359381	0.115245767	0	0.333333333	41	0								
23	1	org.eclipse.ui.workbench/Ec	0.013465611	0.115245767	0	0.333333333	41	0								
24	1	org.eclipse.e4.ui.workbench	0.01337179	0.115245767	0	0.333333333	41	0								
25	1	org.eclipse.core.commands/	0.013361225	0.115245767	0	0.333333333	41	0								
26	1	org.eclipse.ui.workbench/Ec	0.012812006	0.115245767	0	0.333333333	41	0								
27	1	org.eclipse.ui.ide/src/org/ec	0.012674972	0.115245767	0	0.333333333	41	0								

features

Figure 1: Dataset

4 How we solved the problem

A method named DNNLOC is used to find bugs in a program. It uses a system with an input layer, a hidden layer, and an output layer. It takes in several types of information from the bug history and the files that were changed to fix the bug. The system then gives a score that tells how likely it is that a file has a bug.

1) Text Similarity(rVSM): Bug reports are broken down into words. Punctuation and common words are removed. Words are simplified using a standard program. The importance of each word is calculated using a method called Term Frequency - Inverse Document Frequency (tf-idf). These word weights are used as features.

From the source files, terms are extracted in the same way as in bug reports. Most of the source code is removed in this process. The remaining text is mostly the names of things in a source file and the names of API classes and interfaces used in the source file.

The similarity between a bug report and a file is considered as a feature. To get this feature, a method called the revised Vector Space Model (rVSM) is used. The similarity of a bug report B and a file f computed by rVSM is used as the score of this feature for the pair (B, f).

2) Collaborative Filtering Score: This measures how similar a bug report is to previously fixed bug reports by the same file. It is the textual similarity (measured by rVSM) between a bug report B and all the combined texts in the bug reports that were fixed by file f before B was reported.

3) Class Name Similarity: The similarity between the names of classes mentioned in a bug report and those in a source file is also considered. Cosine similarity is used for this calculation.

4) Bug fixing Recency: The bug-fixing recency score is calculated as follows. Let S be the set of bug reports that were filed before bug report B and were fixed in file f. Among S, let B' be the report that was most recently fixed. The bug-fixing recency for a pair of a bug report B and a file f is defined as $(B.month - B'.month + 1)^{-}1$. If f was fixed for a report B' in the same month that B was filed, the value is 1. The further in the past f was last fixed, the smaller the bug-fixing recency score.

5) Bug Fixing Frequency: This feature looks at how many times a source file, let's call it 'f', was fixed. We count how many times 'f' was fixed before the bug report, let's call it 'B', was filed. This count is then used as a feature.

6) DNN Relevancy: Another DNN model is used to handle any mismatch between the bug report and the source file. This DNN takes the tf-idf scores of all the terms that were created for the first feature, which is Text Similarity(rVSM), and gives a score between 0 and 1. This score shows how relevant the bug report and the source file are to each other.

5 EXTRACTING FEATURES

- Since the features for each source code file and bug report pair are not available in the dataset, we had to extract these features. We used a library for text processing. However, we couldn't extract all features because calculating DNN relevancy for all samples takes a lot of time.
- First, we crawled all java files which were identified as buggy in the bug reports and calculated all features for each pair. For example, if 4 different java source files were changed to fix the bug, we produced 4 different samples from this information and labeled each sample as buggy. That means the same bug report becomes a pair with each of these java files and all 5 features are calculated for each pair.
- To train our model, we need bug report and source file pairs which have no relation while fixing the bug reported in the bug report. We went through all bug reports and paired them with 50 irrelevant source files for each buggy java file which belongs to the bug report. That means, if a bug report has 3 different buggy java source files, we calculated features for 150 different irrelevant pairs and labeled them as clean. This increased the number of samples, but it makes the data unbalanced. The ratio of buggy labels and clean labels is 1 to 50.
- We randomly chose other source files in the data set to find irrelevant pairs until we reached enough number of files which are different from the buggy files. We tried to use git in our code script, but we got some inconsistency between the files from the dataset and the files the git checkouts. After that, we decided to stick with just that dataset.
- At the end, we had 445485 bug report and source code file pairs with 5 different features. The number of different samples labeled buggy is 8735, and the number of different samples labeled clean is 436750. We oversampled the data labeled as buggy. After oversampling, the ratio between buggy samples and clean samples became 1 to 5. At the end, we achieved a dataset with length 524100.

6 EXPERIMENT

We utilized a Deep Neural Network (DNN) model with three layers. The model was designed with five input nodes, each representing a different feature: text similarity, collaborative filtering score, class name similarity, bug fixing recency, and bug fixing frequency. The model had a single output node to represent the relevancy score between the bug report and the source file.

We divided our entire dataset into training and testing sets, with an 80:20 ratio respectively. During the training phase, we set the alpha value to 0.00005. The training was halted either after 10000 iterations or after 30 consecutive steps without any change in loss, whichever occurred first.

Subsequently, we conducted experiments to determine the optimal count for hidden nodes that would yield high top-k accuracy and maintain reasonable running times. We tested for hidden node counts ranging from 100 to 1000, but the accuracy values didn't show significant changes. It appeared that even a count of 100 was sufficient for high accuracy. However, we found that using 300 hidden nodes provided a balance between accuracy and training time. Beyond 400, the training time was not efficient enough for iterative testing of our algorithm. Therefore, we settled on 300 for the number of hidden nodes.

After finalizing the hidden node count, we conducted a series of 10-fold experiments on the training dataset. The reported accuracies are the average accuracies obtained from these 10-fold experiments. This approach ensured a robust and reliable evaluation of our model's performance.

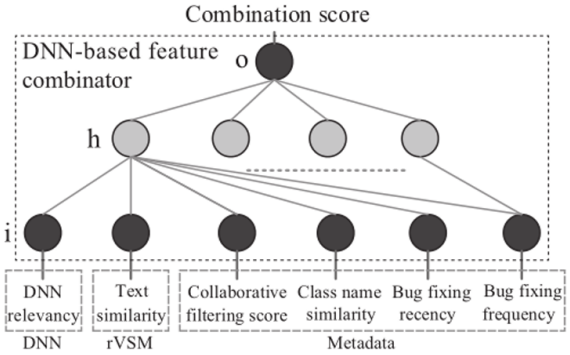


Figure 2: DNNLOC Bug Localization

7 CONCLUSION

The study discusses a comparative analysis of two bug localization studies, one focusing on information retrieval (IR) and the other on deep learning approaches. In the initial study, described in a previous paper, bug localization relied exclusively on the rVSM (relevance Vector Space Model) score, an improved cosine similarity metric. Conversely, the subsequent study introduced in this paragraph incorporated the rVSM feature alongside additional textual features through a deep neural network, aiming to enhance accuracy.

The effectiveness of both deep learning and information retrieval techniques in bug localization is highlighted, with the implementation also resulting in a significant increase in top-k accuracy. It presumably provides a comparative overview of top-k accuracies from both papers and their respective implementations. While the results demonstrate considerable similarities with the original values, minor discrepancies are noted, attributed primarily to the absence of the DNN Relevancy Score in the current implementation.

In essence, the paragraph underscores the detailed examination of the implementation specifics of two bug localization studies. The first study adopted an information retrieval approach, relying solely on the rVSM method, while the second study integrated the rVSM feature with other textual features within a neural network framework, yielding superior top-k accuracy.

7.1 Technical Information

For bug localization in Python, you can use various technical libraries and tools. Some of the commonly used ones include:

Machine Learning Libraries:

- NLTK (Natural Language Toolkit)
- Scikit-learn
- TensorFlow
- Keras

Version Control System APIs:

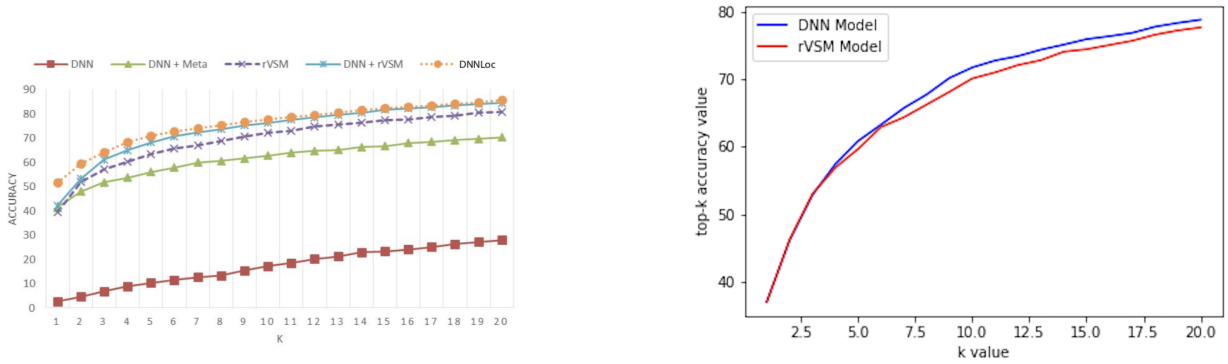
- - GitPython (for working with Git repositories)

Text Embedding Models:

- Word2Vec
- Doc2Vec
- BERT (using Hugging Face's transformers library)

Other Technical Requirements:

- Dataset of bug reports and associated source code
- Preprocessing techniques for bug reports and source code
- Feature extraction methods
- Machine learning or deep learning models for bug localization
- Evaluation metrics (e.g., precision, recall, F1-score)
- Cross-validation techniques
- Documentation and reporting tools



(a) Original Results

(b) Our Results

Figure 3: Comparison of Results

7.2 What new things we learned

Creating a bug localization project using a Deep Neural Network (DNN) involves several new learnings:
Deep Learning Fundamentals:
Understanding the basics of deep learning, including neural network architecture, activation functions, loss functions, and optimization algorithms. Implementing deep learning models using Python and frameworks like TensorFlow or PyTorch.
Text Preprocessing Techniques:
Preprocessing bug reports and source code for input into the deep learning model. Techniques such as tokenization, stop-word removal, stemming or lemmatization, and vectorization.
Sequence Models for Text:
Understanding how to represent text data for deep learning models, such as using word embeddings like Word2Vec, GloVe, or training your own embeddings. Implementing recurrent neural networks (RNNs) or convolutional neural networks (CNNs) for sequence modeling.
Hyperparameter Tuning:
Tuning hyperparameters of deep learning models for better performance. Understanding the impact of different hyperparameters on the model's performance and training time. Model Evaluation and Interpretation: Evaluating the performance of the deep learning model using appropriate metrics such as precision, recall, F1-score, and accuracy. Interpreting the predictions made by the deep learning model to understand which features contribute most to bug localization.
Deployment Considerations:
Considering deployment aspects such as model size, inference time, and memory requirements. Exploring options for deploying deep learning models in production, such as using frameworks like TensorFlow Serving or converting models to formats suitable for deployment.
Project Management Skills:
Managing a complete project lifecycle, including data collection, preprocessing, model development, evaluation, and reporting. Collaborating with team members, if applicable, and utilizing version control systems like Git.

7.3 Results Screenshots

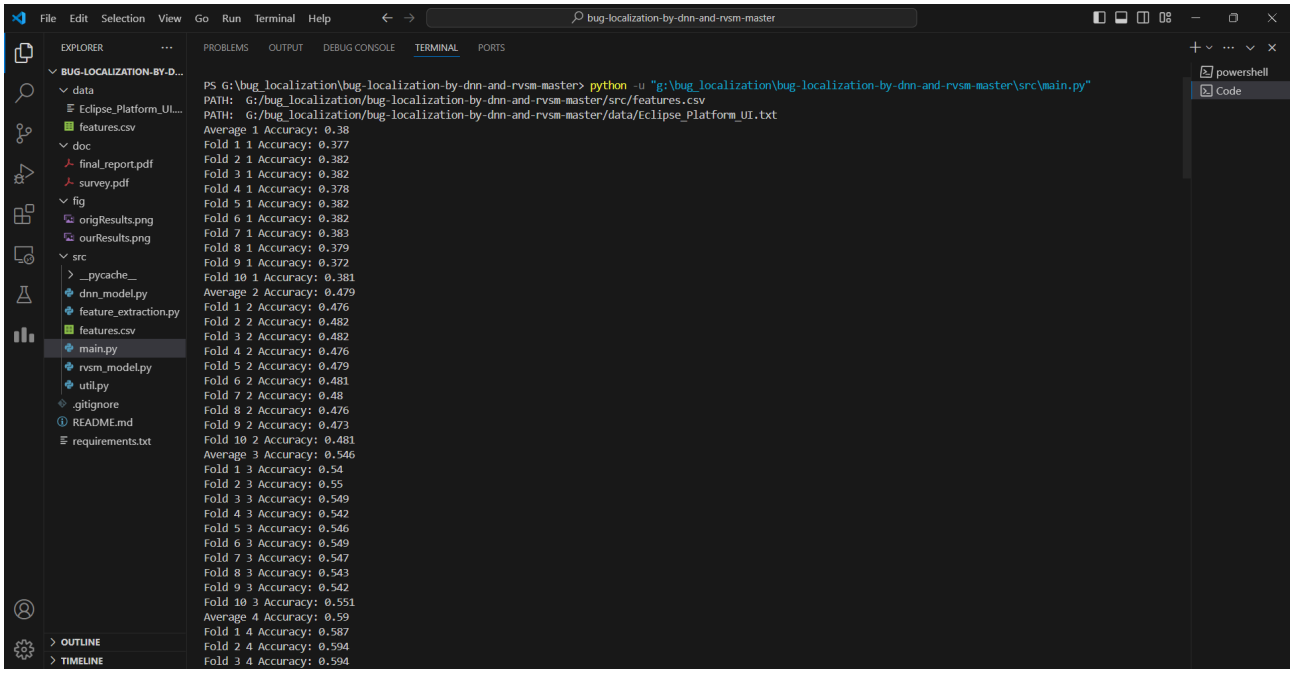


Figure 4: ScreenShot-1

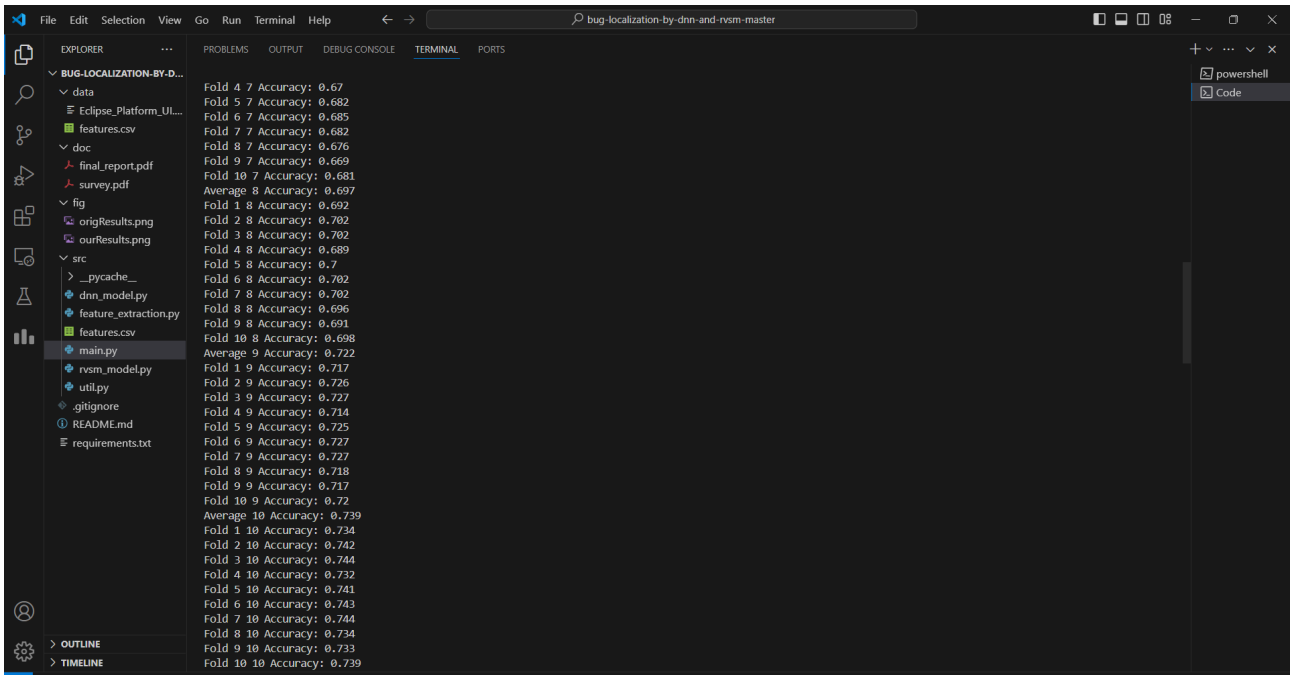


Figure 5: ScreenShot-2

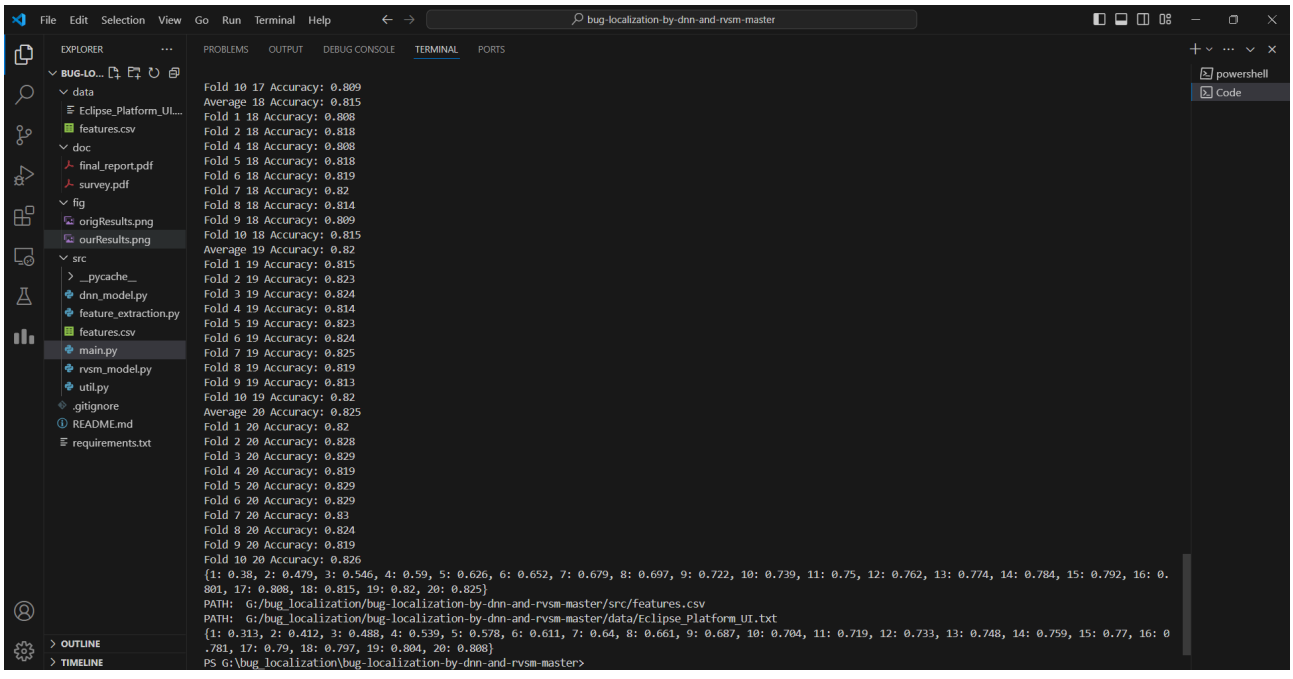


Figure 6: ScreenShot-3