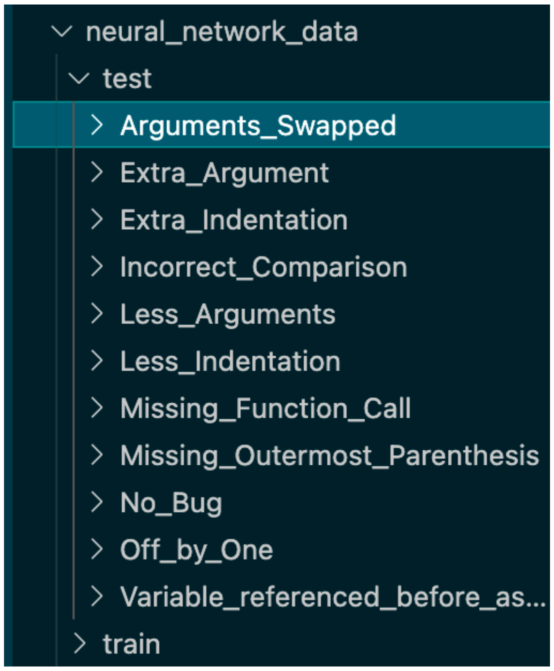
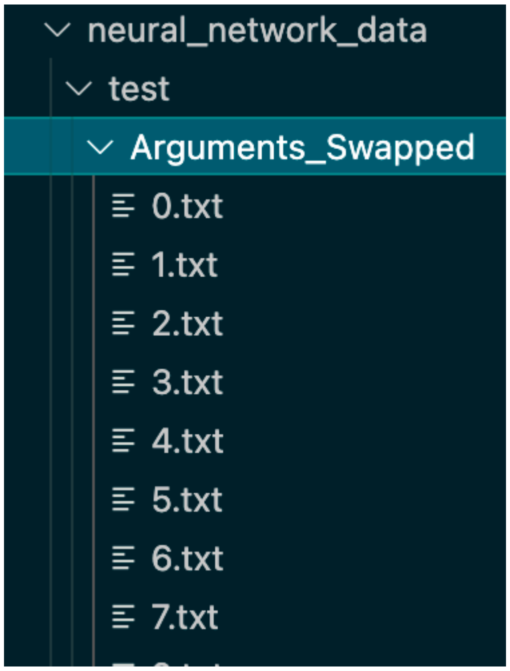
1. **Changes**: No major changes have been made to the novelty, user community, dataset(s) or comparison subjects from the first progress report. Any changes in scope of the project are minor since they are purely-implementation based and can be found in the 'Challenges' section of this document.
2. **Research Questions**: Following are the primary research questions that I believe my work will answer:
   - **(I)** How well do ensemble methods from Scikit-learn do when attempting to classify 'bug-free' data vs. 'buggy' data in Python code when
     - **a)** Multiple bug-types need to be detected and classified
     - **b)** The bugs are artificially introduced
   - **(II)** How well do neural networks from Tensorflow do when attempting to classify 'bug-free' data vs. 'buggy' data in Python code when
     - **a)** Multiple bug-types need to be detected and classified
     - **b)** The bugs are artificially introduced
   - **(III)** Can classification accuracy or training time of the aforementioned algorithms be substantially improved based on the feature vectorization chosen to convert code into ML-consumable vectors?
   - **(IV)** Given the problem being solved, which ML method performs the best and how do they compare against each other?
   - **(V)** Is the performance of the best classifier comparable to industry and research standards? Are there any takeaways from the work done that could be applied to future research in the field?
3. **Challenges**: Following are the primary challenges I faced while composing my final project and how I either solved or avoided them:
   - Given the wide variety of code being ingested and then processed to contain an artificially-induced bug, the first problem I faced was ensuring that
     - ✓ In-line comments or block comments were ignored successfully
     - ✓ Bugs were introduced in the correct places, eg: argument swapping bugs must not be introduced in function definitions but only in function calls
     - ✓ Bugs that were introduced at a specific point in the line of code, did not affect the rest of the line - which would otherwise harm classification
     - ✓ Any type of Python line could be extracted, processed and dealt with accordingly
     - ✓ Bugs of each type were introduced at all possible locations ensuring maximum variety of training data

     I solved the above issues by ensuring that each of the functions generating bugs in extracted code were robust by testing them on a variety of edge cases to ensure desired performance. In addition, if a bug cannot be generated in a line by one of the functions, the attempted 'buggy' line is not added to the training data. Furthermore, I added conditionals in the parsing code that ensure that each extracted line is a valid, correctly-formatted line of code before adding it to the training dataset or artificially generating bugs in said line.
   - Using the 'ast' module proved to make vectorization a more complicated problem - which if not solved correctly would adversely affect classification accuracy.
     - ✳ My original attempt for classification involved passing in two inputs to the classifier: 1) the line of code being examined and 2) the prior lines of code within the same function. This was proving to be a complicated problem to solve especially given the perceived gain in accuracy at this stage of building the tool. My code is currently set up to extract the prior lines of code within a function up until the line being examined however I am not utilizing the same as of right now. This functionality may be revisited at a later date - either before or after the project deadline.

- ✳ Given that the classifier would now only be using one line at a time as input, the 'ast' module was no longer necessary to break up the line into its constituent parts. It was also throwing errors for most lines (since it requires a 'functionally-complete' piece of code to analyze). Last of all, vectorizing an 'ast' representation of a line was more complicated than vectorizing the line itself and instead of improving feature recognition was harming it.
- ✳ In light of the above, each line is now being vectorized directly with either scikit-learn's CountVectorizer or TFIDFVectorizer or tensorflow's TextVectorizer. Despite their intended use for NLP, all the aforementioned vectorizers are able to extract and preserve the necessary syntax and programming features from the code. Besides, these vectorizers having been built to be used with their respective classifiers make classification more efficient and accurate.

- The original input pipeline involving the usage of a .csv file to store the line and its label that could then be easily converted into a pandas data frame was proving to be slow and inefficient when training the Neural Network - and was likely having an adverse affect on accuracy as well.
    - ✳ To this end, I added a second input pipeline to the tool that involves using tensorflow's **preprocessing.text_dataset_from_directory()**. The pipeline now saves each extracted and processed line to a .txt file that contains all the lines from the same file. Based on the type of line - 'No bug', 'Arguments swapped', 'Extra Argument', etc., the line is added to .txt file with the same name but in a folder corresponding to its label. The folder structure therefore looks as follows:


```
∨ neural_network_data
  ∨ test
    > Arguments_Swapped
    > Extra_Argument
    > Extra_Indentation
    > Incorrect_Comparison
    > Less_Arguments
    > Less_Indentation
    > Missing_Function_Call
    > Missing_Outermost_Parenthesis
    > No_Bug
    > Off_by_One
    > Variable_referenced_before_as...
  > train
```


```
∨ neural_network_data
  ∨ test
    ∨ Arguments_Swapped
        0.txt
        1.txt
        2.txt
        3.txt
        4.txt
        5.txt
        6.txt
        7.txt
```

- ✳ This input pipeline now allows users to train/test the tool on pre-formatted data of two different structures increasing the versatility of the tool.
- ✳ Most importantly, utilizing this input pipeline allowed me to leverage other preprocessing tools native to tensorflow datasets such as caching and prefetching the data both of which exhibited massive speedups per epoch

        when the neural network was used. This allowed for quicker turn around time in training and testing of the neural network as well as easier usage of the TextVectorization layer in tensor flow. Other preprocessing options native to tensorflow can now also be explored - time permitting.

- Optimizing the structure of the Neural Network was becoming non-intuitive and therefore time-consuming to no avail.
    - ✳ I solved this issue by making the Neural Network model buildable by a function with certain hyper-parameters that controlled features such as the type of layers, type of activation functions, number of neutrons per layer, etc. This function with its hyper-parameters could then be passed into Keras' Hyperband tuner that could run the model multiple times on different combinations of the hyper-parameters to locate the optimal combination faster than I would be able to by hand.
    - ✳ This allowed me greater flexibility in the amount of hyper-parameters in the Neural Network that I could tune and optimize and showed great increases in accuracy. Also, to the best of my knowledge, this approach has not been commonly (if at all) taken in research relating to bug detection using Neural Networks yet and so adds another layer of novelty to the tool.
    - ✳ This process was also greatly helped by the preprocessing enabled by the second pipeline.

4. **Additional Details**: Below are additional details surrounding my project:
    - Once both the database of Python repositories I am using as my input data as well as my own codebase are finalized, I will be uploading all of the above in a well-organized structure to a Github repository which will be accessible to the course staff.
    - I will be using Jupyter notebooks for my own codebase. The notebooks will contain the results from previous runs so that a user need not run the entire training or test suite themselves to see the results generated. In addition, Jupyter notebooks offer Markdown cells that are better suited for periodic explanation of results and code than in-line code comments.
    - If a user would like to run the code themselves, the Github repository will be set up to support any of the Jupyter notebooks being run from start to finish within the repository once it is downloaded to a local machine.
    - The open-source Python modules I will be using are primarily (although not the complete list) Tensorflow, Scikit-learn, Numpy, Matplotlib, Pandas and Keras.