CropSegmentation: M3

1. M2 corrections

After the feedback and instructions received during the presentation of the second milestone:

- The best model was added to the MLflow model registry;
- We received instructions to modify the naming convention of the saved model files in the dvc.yaml file, and we are still working on finding a solution.

2. Quality Assurance with Pytest

Pytest was used to perform Model Training Testing and Behavioral Testing.

1. **Model Training Testing: Ensured that the model trains successfully and is able to run on various devices.**
    1.1. **Checking Decreasing Loss Post-training on a Batch:** Validated if there's a decrease in loss after training on one batch by implementing `test_loss_decrease.py`. In this script, the `test_decreasing_loss` function runs two training epochs and asserts that the loss after the second epoch is less than the loss after the first epoch. This is to ensure that the model is learning and improving its performance over time.
    1.2. **Checking Overfit on a Batch:** The model was trained on a single batch for several epochs, after which we assessed if the accuracy fell within a specific range. This served as an indicator of the model fitting that particular batch. This test was implemented in the `test_overfit_batch.py` file.
    1.3. **Training Completion Verification:** Verified if the training process completes. This was done by implementing the following assertions in the

`test_training_completion.py` file: asserting that the training has likely completed by asserting that the loss history is not empty (implying training iterations occurred) and checking that the model file has been saved.

1.4. **Device Compatibility Check during Training:** Ensured that the training process is successful on different devices, including CPU, CUDA, and MPS (MacOS backend for GPU training acceleration). This was done by implementing the test_device_training.py script, which first runs the training on a CPU and asserts that the loss history is not empty, indicating that training has occurred. If CUDA is available, it runs the training on CUDA and asserts that the loss history is not empty. Lastly, if MPS is available, it runs the training on MPS and asserts that the loss history is not empty.

To perform the above testing, several modifications were made in `train_model.py` to make the code more modular and testable:

1. **Loss Decrease Testing Refactoring:** The `train` function was modified to return the list of training and validation loss values. Additionally, the training logic was encapsulated in a separate function called `run_training_epoch` for easier testing.
2. **Overfit Testing Refactoring:** Added the function `train_single_batch` to train the model on a single batch.
3. **Training Completion Testing Refactoring:** The `train` function was modified to return the final learning rate and the saved model's path.
4. **Device Testing Refactoring:** The `train` function was altered to accept the device as a parameter.

2. Behavioral Testing: Evaluated the model's behavior and response to different inputs and conditions.

2.1. **Invariance testing:** Tested if changes in the input, such as flipping an image, do not significantly affect the output. The test is performed through the `test_invariance.py` script, in which for each image in the test set, a prediction is made on the original image. The image is then flipped vertically,

and a new prediction is made. The predictions of the original and flipped images are then compared through their Jaccard scores. The test is flagged unsuccessful if the prediction differences surpass a predefined threshold.

2.2. **Directional testing:** Assessed the model's sensitivity to controlled modifications in the input. More specifically, the `test_directional.py` script tests if the model's predictions reflect modifications in input images. It adds a rectangle to each test image, makes predictions for the original and modified images, and checks if Jaccard scores are within a tolerance of 0.2. If not, it raises an assertion error.

2.3. **Minimum functionality testing:** The goal of this test is to examine the model's behavior under both optimal and challenging conditions. Two sets of images were manually selected from the testing data, one set representing optimal conditions, where the model was expected to perform well, while the other represented more challenging conditions to test the model's robustness and ability to generalize. In the `test_minimum_functionality.py` script, the `test_optimal_conditions` and `test_challenging_conditions` functions were implemented. Each function loads the model and iterates through its respective set of images, making predictions, and computing the Jaccard score against the true labels. The mean Jaccard score across all images in each set is then compared to a predetermined threshold to ensure the model's satisfactory performance under both favorable and challenging conditions, thus verifying the model's robustness and ability to generalize across different scenarios.

To maintain code organization and readability:

- A `tests` folder was introduced in the project repository.
- Inside the `tests` folder, sub-folders `test_model_training` and `behavioral_testing` were created to contain the respective tests.
- The project organization within the README was updated accordingly.

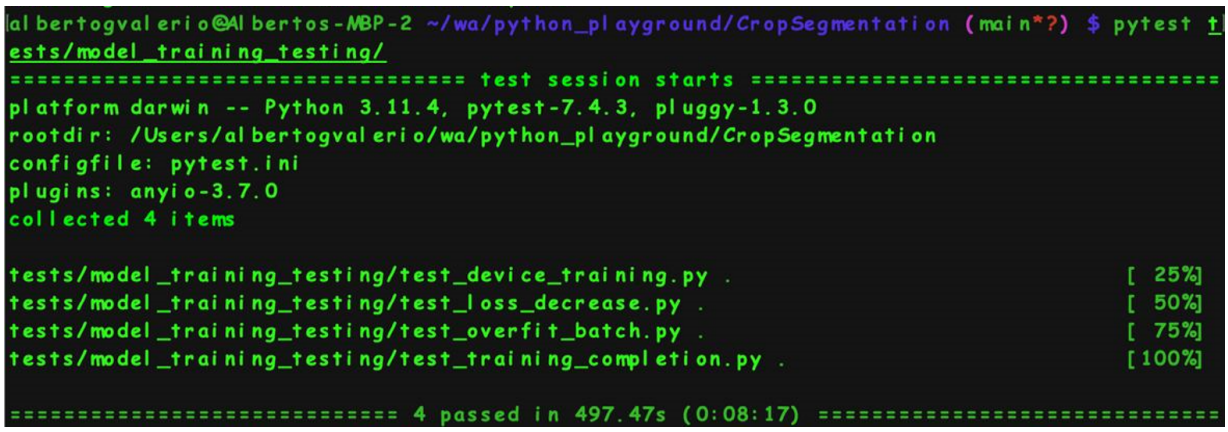
3. Additional Testing of Utility Functions

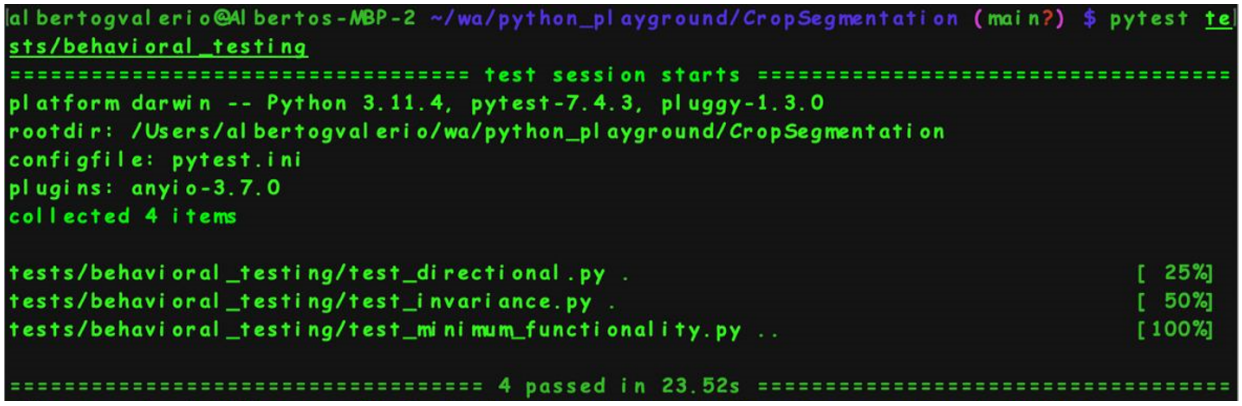
Although not explicitly requested by this milestone, we decided to perform additional quality assurance by using Pytest to also test some of the most important utility functions of our project. This allowed us to guarantee that these functions behaved as expected and reliably executed their intended tasks. Here is a breakdown of these tests:

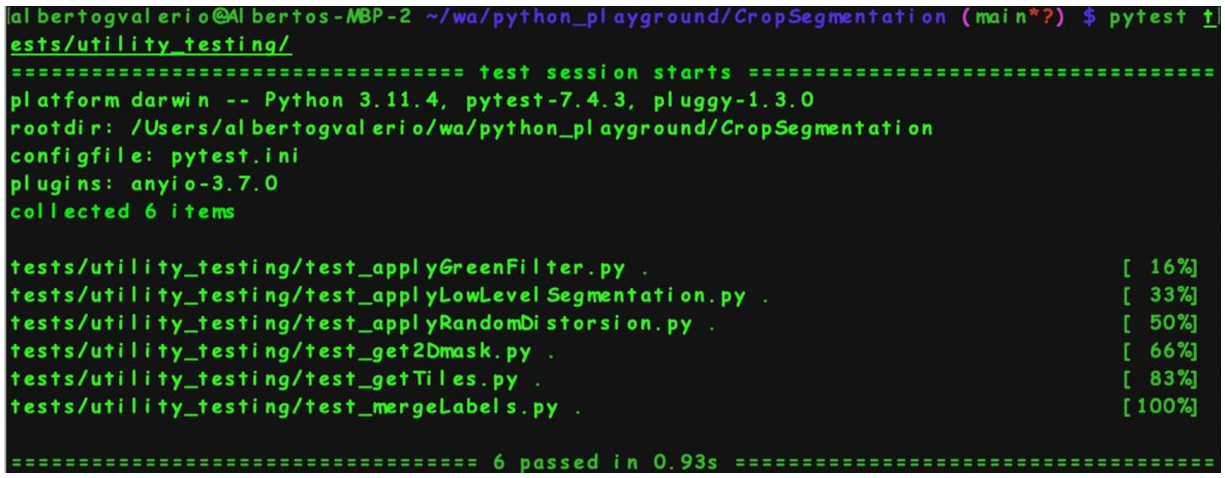
1. **Test for Green Filter Application:** This test, implemented in test_applyGreenFilter.py, verifies the application of a green filter to an image. Specifically, it creates a random 100x100 RGB image, applies the green filter to this image, and asserts that the shape of the filtered image matches the input image's shape and that the values of the filtered image lie within the [0, 255] range, ensuring the filter operation's correctness.
2. **Test for Low Level Segmentation:** This test ensures that the applyLowLevelSegmentation function returns the correct number of segmented images for both clustering and thresholding algorithms. This is done by creating a set of dummy images, applying low-level segmentation using the clustering and thresholding algorithms and asserting that the function returns the expected number of segmented images for both these algorithms.
3. **Test for Random Distortion Application:** This test generates dummy images and masks, applies random distortions to them, and then asserts that the number of distorted images and masks matches the number of original dummy images and masks, ensuring the distortion function is executed correctly.
4. **Test for 2D Mask Generation:** This test produces a random RGB mask, converts this mask to a binary 2D mask, and then verifies that the resulting mask's shape is as expected and that it only contains values of either 0 or 255.
5. **Tile Extraction Test:** This test creates 100x100 RGB dummy images, extracts tiles of sizes 10x10 and 20x20 from this image, and validates that the number of extracted tiles and their sizes match the expected values.

6. **Test for Label Merging:** This test generates a dummy mask with a specific pattern, merges the labels within this mask, and ensures that the merged mask has the correct shape and value distribution.

Below are screenshots showcasing the successful execution of all the implemented tests:





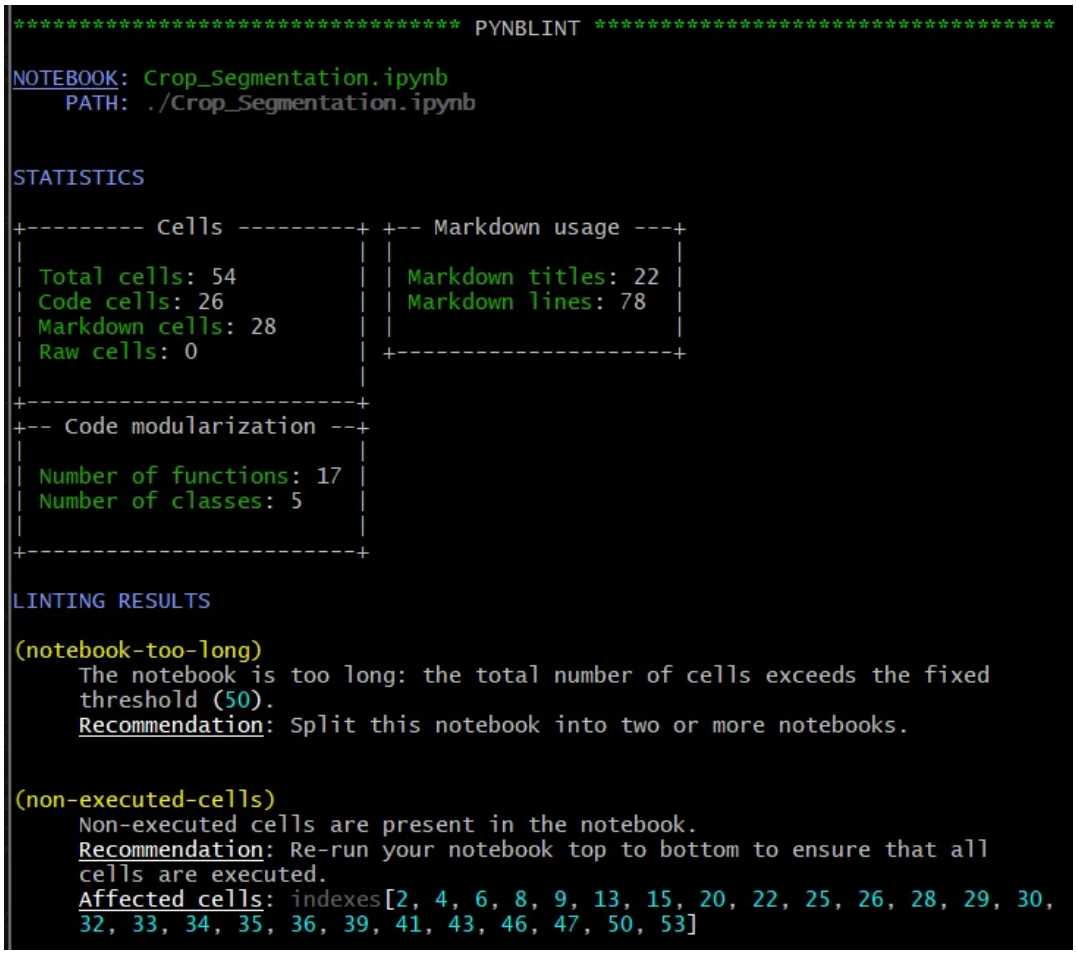


3. Code Analysis with Pynblint

Pynblint was used to perform a set of checks on the project notebooks based on best practices that have been empirically validated; upon identifying potential defects, Pynblint offered recommendations to enhance the notebooks' quality.

In particular, it was used to analyze the notebook related to the initial exploration of the data. The first Pynblint analysis identified the following issues, as can be seen in the screenshots provided below:

- ***notebook-too-long***: the notebook was too long, the total number of cells exceeding the fixed threshold of 50 cells;
- ***non-executed-cells***: the notebook presented non-executed cells;
- ***cell-too-long***: the cells were too long: they exceeded the fixed threshold of 30 lines.



```
(cell-too-long)
     One or more code cells in this notebook are too long (i.e., they exceed
     the fixed threshold of 30 lines).
     Recommendation: Consider consolidating your code outside the notebook by
     moving utility functions to a structured and tested codebase.
     Use notebooks to display results, not to compute them.
     Affected cells: indexes[2, 6, 13, 22]
```

Given the obtained recommendations, we addressed the alerts by modifying the notebook. The ***notebook-too-long*** alert was solved by merging the content of some cells so that the total number of cells didn't exceed the fixed threshold. In order to address the ***non-executed-cells*** alert, we ran the notebook from the beginning to ensure that all cells were executed. As for the ***cell-too-long*** alert, we divided the content of the cells into smaller, more manageable sections, thus resolving the issue.
Then, another Pynblint analysis was run on the modified notebook, resulting in the following alerts, as shown in the screenshot below:

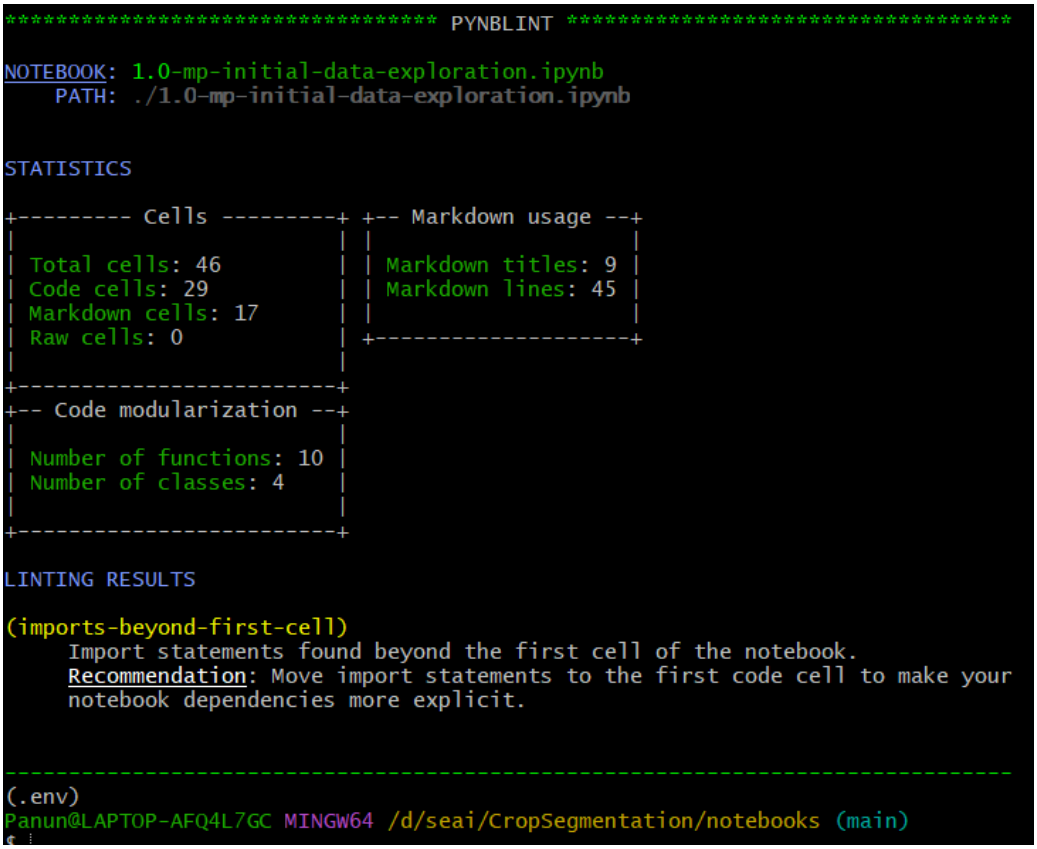
- ***non-linear-execution***: notebook cells have been executed in a non-linear order;
- ***cell-too-long***: the cells are too long: they exceed the fixed threshold of 30 lines.

```
********************************** PYNBLINT ***********************************

NOTEBOOK: 1.0-mp-initial-data-exploration.ipynb
    PATH: ./1.0-mp-initial-data-exploration.ipynb


STATISTICS

+--------- Cells ---------+ +-- Markdown usage --+
|                          | |                    |
| Total cells: 32          | | Markdown titles: 9 |
| Code cells: 15           | | Markdown lines: 45 |
| Markdown cells: 17       | |                    |
| Raw cells: 0             | +--------------------+
|                          |
+--------------------------+
+-- Code modularization --+
|                         |
| Number of functions: 10 |
| Number of classes: 4    |
|                         |
+-------------------------+

LINTING RESULTS

(non-linear-execution)
     Notebook cells have been executed in a non-linear order.
     Recommendation: Re-run your notebook top to bottom to ensure it is
     reproducible.


(cell-too-long)
     One or more code cells in this notebook are too long (i.e., they exceed
     the fixed threshold of 30 lines).
     Recommendation: Consider consolidating your code outside the notebook by
     moving utility functions to a structured and tested codebase.
     Use notebooks to display results, not to compute them.
     Affected cells: indexes[2, 4, 12, 23, 26]
```

In light of these results, the notebook was once again modified to align with these new recommendations. Specifically, the ***non-linear-execution*** alert was solved by re-running the cells in the notebooks in the correct order. On the other hand, the ***cell-too-long*** alert was accepted as a compromise to prevent triggering the ***notebook-too-long*** alert again, which could cause a loop in corrections.

Following this, we ran one last Pynblint analysis on the modified notebook, which resulted in the following alert, as can be seen in the screenshot below:

- ***import-beyond-first-cell***: import statements found beyond the first cell of the notebook.



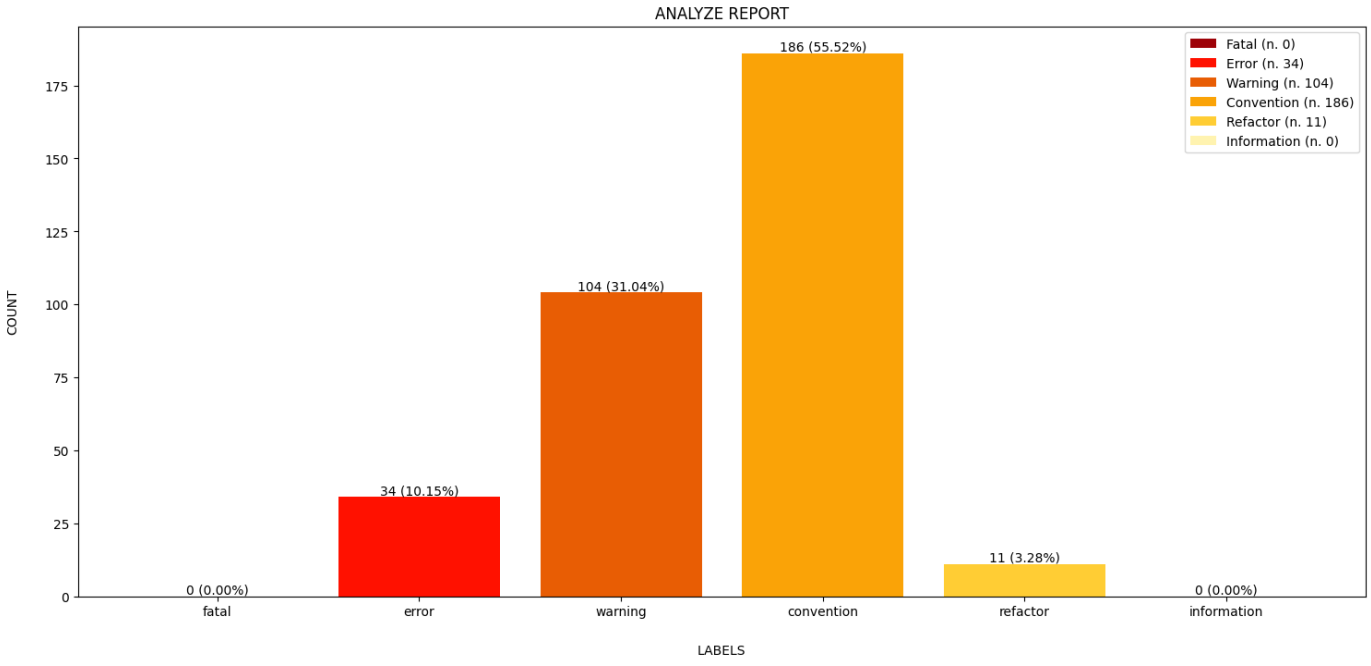
Addressing this issue would have meant moving all the imports into the first cell. However, the first cell containing all the imports had been purposely divided into more cells in order to solve the previously encountered ***cell-too-long*** alert; therefore, this alert was disregarded and the code analysis with Pynblint was considered completed.

By implementing the empirically validated best practices recommended by Pynblint based on the issues detected in our project notebook, we were able to improve the quality of our code.

4. Code Analysis with Pylint

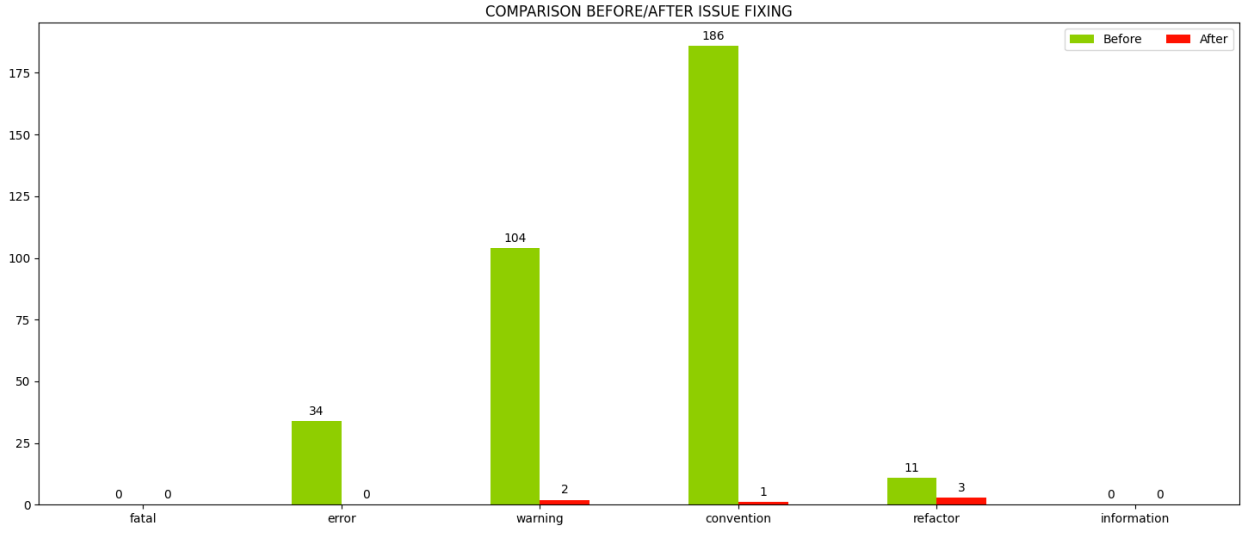
The project's source code was analyzed with Pylint, a static code analysis tool. This tool scans the code without executing it, thus providing an early detection mechanism for potential issues. The initial scan detected 335 issues categorized into four distinct categories: error, warning, convention, and refactor. While Pylint has two additional categories—'fatal' and 'information'—none of these were detected within our project's codebase.

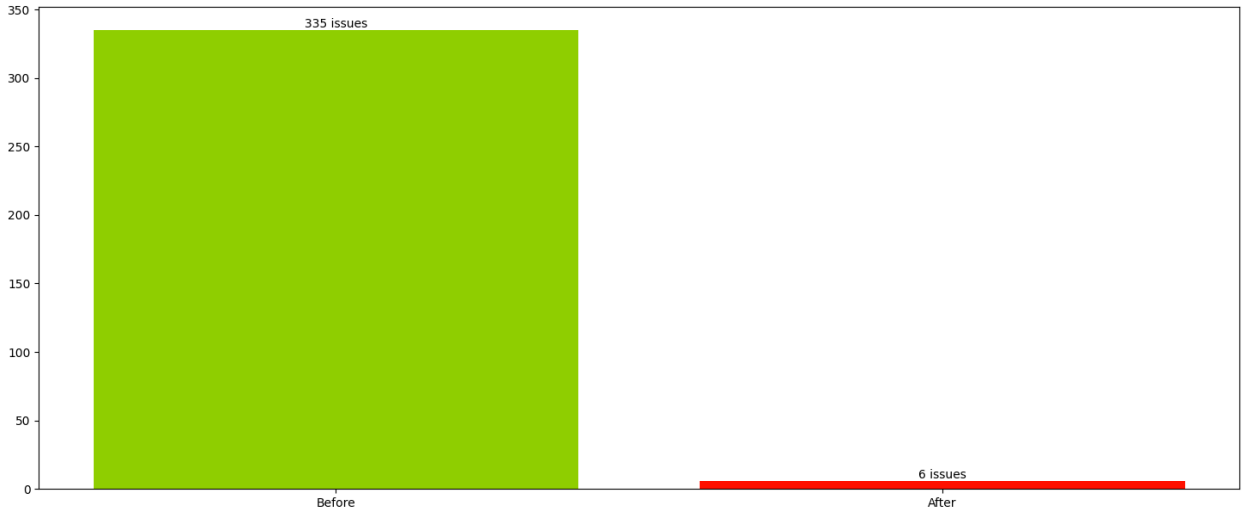
Below is a representation illustrating the distribution of the detected issues among the different categories:



The resolution strategy involved addressing the issues in a hierarchical manner, starting with the most severe ones. This approach enabled a systematic resolution, ensuring that critical issues were addressed as soon as possible. Within each category, the most frequent issues were prioritized, to ensure a thorough and efficient resolution process.

After the issue resolution, a follow-up scan was conducted using Pylint. This subsequent analysis detected only six issues. After careful examination, these issues were determined to be false positives and were consequently left unresolved. The comparative results, pre and post-resolution, are depicted in the graph below, showcasing the significant improvement in code quality.

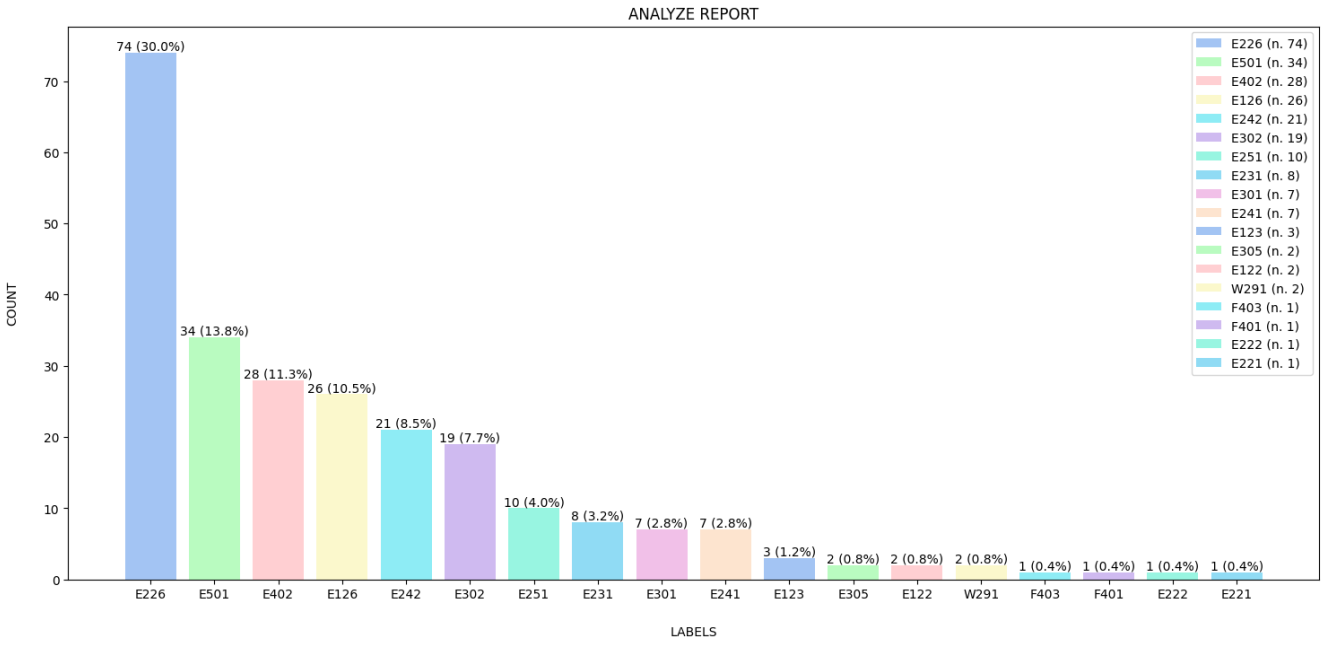




5. Code Analysis with Flake8

After the Pylint analysis, the project's code was further analyzed using Flake8. Unlike Pylint, Flake8 does not categorize issues based on severity; instead, it accumulates the results for each file analyzed. Given that Flake8 was executed following the Pylint analysis, it is possible that a significant number of issues were already addressed, hence not flagged during this stage.

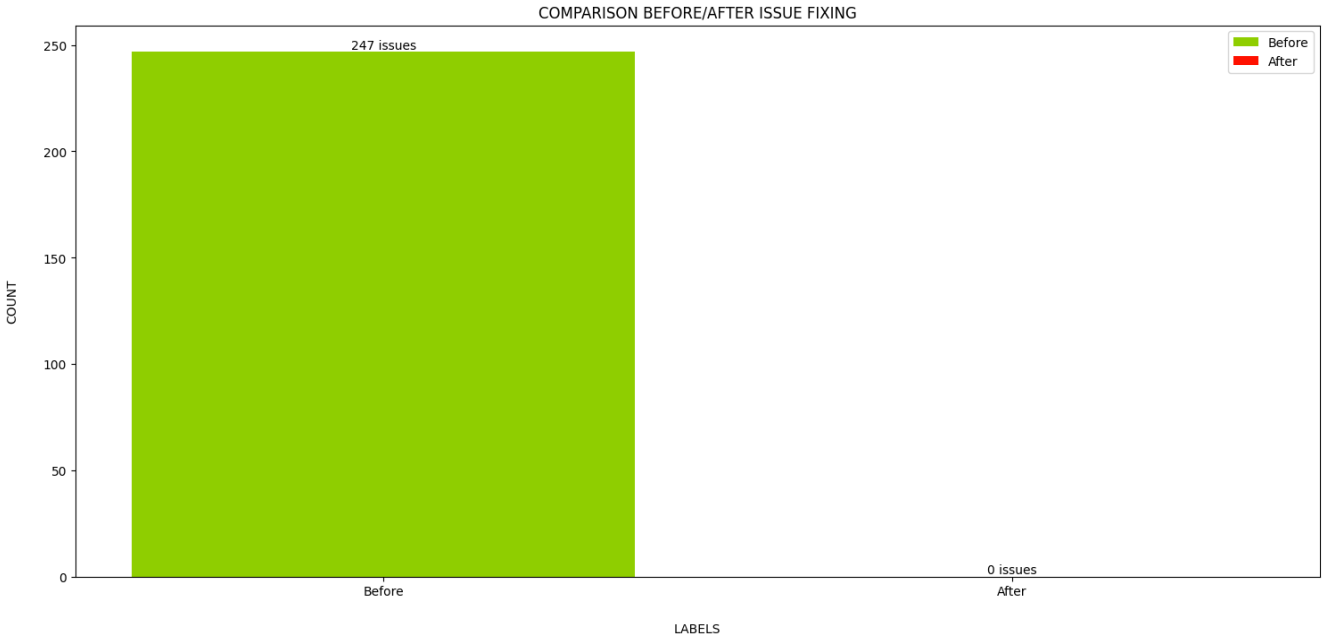
The initial scan with Flake8 flagged 247 issues diversified into 18 types, as demonstrated in the graph below:



The primary issues detected revolved around missing whitespace around arithmetic operators, over-indentation for hanging indents, and tab inconsistencies. These issues generally pertained to the formatting and structuring of the code, aligning with Flake8's core objective of enforcing style consistency and code simplicity.

Following the identification of these issues, a systematic resolution approach was adopted, and a re-scan was performed post-resolution. The follow-up scan revealed a remarkable

improvement with zero issues detected. The comparison of the code quality before and after issue resolution is pictorially represented in the graph below:



Both for Pylint and Flake8, configuration files were defined, named .pylintrc and .flake8 respectively, to guide the scanning process. These configuration files defined general settings that the tools should follow during the scans, for instance, overlooking certain checks deemed irrelevant to our project, and specifying the paths of the imported modules among others.

6. Quality Assurance with Deepchecks

Quality Assurance with Deepchecks was carried out to ensure that our model is reliable and accurate in the task of semantic segmentation.

1. **Implementation**
   Before running the Deepchecks suites, the labels and predictions of our data were converted to a predefined format that is required by the Deepchecks suites. This was done using a standard DataLoader object and a newly defined collate function that converted the batch to the required format. The implementation of this function is shown in the screenshot below:

```python
def deepchecks_collate_fn(batch) -> BatchOutputFormat:

    # batch received as iterable of tuples of (image, label) and transformed to tuple of iterables of images and labels:
    batch = tuple(zip(*batch))

    # images:
    images = [(tensor.numpy().transpose((1, 2, 0)) * 255).astype(np.uint8) for tensor in batch[0]]

    #labels:
    labels = [mask.type(torch.int8) for mask in batch[1]]

    #predictions:
    predictions = [make_predictions(model, img) for img in images]
    predictions = [convert_prediction(pred) for pred in predictions]

    return BatchOutputFormat(images=images, labels=labels, predictions=predictions)
```

   Using the `DataLoader` objects, we instantiated the `VisionData` class, which is the base class for storing data for vision tasks; this allowed Deepchecks to run its suite of functionalities on the data by wrapping batches.

```python
train_loader = DataLoader(dataset=train_dataset, shuffle=True, collate_fn=deepchecks_collate_fn)
test_loader = DataLoader(dataset=test_dataset, shuffle=True, collate_fn=deepchecks_collate_fn)

training_data = VisionData(batch_loader=train_loader, task_type='semantic_segmentation', label_map=LABEL_MAP)
test_data = VisionData(batch_loader=test_loader, task_type='semantic_segmentation', label_map=LABEL_MAP)
```

   By invoking a suite, Deepcheck performs all the checks that are relevant to the specific task type. In our case, the task type is semantic segmentation.

2. **Vision Properties**

   Properties are one-dimension values that are extracted from either the images, labels or predictions; they are used by some of the Deepchecks checks in order to extract meaningful features from the data, since some computations are difficult to compute directly on the images. Inspecting the distribution of the property's values can help

uncover potential problems in the way that the datasets were built, or hint about the model's expected performance on unseen data.

- **Image Properties**
  - **Aspect Ratio:** ratio between height and width of image (`height/width`);
  - **Area**: area of image in pixels (`height*width`);
  - **Brightness:** average intensity of image pixels. Color channels have different weights according to RGB-to-Grayscale formula;
  - **RMS Contrast:** contrast of the image, calculated by standard deviation of pixels;
  - **Mean Red Relative Intensity:** mean over all pixels of the red channel, scaled to their relative intensity in comparison to the other channels `[r/(r+g+b)]`;
  - **Mean Green Relative Intensity:** mean over all pixels of the green channel, scaled to their relative intensity in comparison to the other channels `[g/(r+g+b)]`;
  - **Mean Blue Relative Intensity:** mean over all pixels of the green channel, scaled to their relative intensity in comparison to the other channels `[b/(r+g+b)]`.
- **Label & Prediction Properties**
  - **Samples per Class:** This property refers to the number of instances or samples available for each class in the dataset. In the context of semantic segmentation, a class refers to a specific category that each pixel in an image can be labeled as;
  - **Segment Area:** This property refers to the area occupied by each segment in the segmented image. In semantic segmentation, each pixel in an image is assigned a class, effectively dividing the image into different segments. The segment area can provide insights into the size and distribution of different classes in the image;
  - **Number of Classes per Image:** This property refers to the number of distinct classes present in each image.

3. **Suites**

- **Data Integrity: this suite is employed to perform validation on data, whether it's on a batch or right before splitting it or using it for training.**
  - **Image Property Outliers:** find outliers images with respect to the given image properties;
  - **Label Property Outliers:** find outliers labels with respect to the given label properties.

- **Train Test Validation: this suite is used to compare the distribution between train and test datasets.**
  We introduce the *drift score*, which measures the change in the distribution of data over time, and it is also one of the top reasons why machine learning model's performance degrades over time. The drift score is a measure for the difference between two distributions.

    - **Label Drift:** calculate label drift between train dataset and test dataset, using statistical measures;
    - **Image Property Drift:** calculate drift between train dataset and test dataset per image property, using statistical measures;
    - **Heatmap Comparison:** check if the average image brightness is similar between train and test set
    - **Image Dataset Drift:** calculate drift between the entire train and test datasets (based on image properties) using a trained model

- **Model Evaluation: this suite provides checks against a trained model.**
  We introduce the *Dice similarity coefficient*, an estimation of the similarity of two samples. It is different from the Jaccard index which only counts true positives once in both the numerator and denominator; DSC is the quotient of similarity and ranges between 0 and 1. It can be viewed as a similarity measure over sets.

    - **Class Performance:** summarize given metrics on a dataset and model;
    - **Prediction Drift:** calculate prediction drift between train dataset and test dataset, using statistical measures;
    - **Weak Segments Performance:** using image properties, search for model's weak segments with low performance scores.