Software Engineering for AI-Enabled Systems

A.A. 2023-2024

# Crop Segmentation: A Case Study

*Authors:*

Eleonora Ghizzota

Mariangela Panunzio

Katya Trufanova

Alberto G. Valerio

*Supervisors:*

Professor Filippo Lanubile

Research Fellow Luigi Quaranta

PhD. Student Andrea Basile

December, 2023

# Contents


**1 Introduction** **3**

**2 Inception** **3**
- 2.1 Repository Creation . . . 3
- 2.2 Model and Dataset Card Development . . . 3

**3 Reproducibility** **4**
- 3.1 Corrections . . . 4
- 3.2 Data Versioning with DVC . . . 4
- 3.3 MLflow Integration . . . 5
- 3.4 DagsHub . . . 5

**4 Quality Assurance** **6**
- 4.1 Quality Assurance with Pytest . . . 6
- 4.2 Code Analysis with Pynblint . . . 8
- 4.3 Code Analysis with Pylint . . . 12
- 4.4 Code Analysis with Flake8 . . . 13
- 4.5 Quality Assurance with Deepchecks . . . 15
  - 4.5.1 Vision Properties . . . 15
  - 4.5.2 Suites . . . 16

**5 API** **17**
- 5.1 Corrections . . . 17
- 5.2 API Endpoint Implementation with FastAPI . . . 17
- 5.3 Ensuring API Robustness with Pydantic . . . 19
- 5.4 API Documentation with ReDoc . . . 19
- 5.5 Quality Assurance with Pytest for API Endpoints . . . 20
- 5.6 Showcasing API Capabilities through an Interactive Web Application . . . 22

**6 Deployment** **24**
- 6.1 Corrections . . . 24
  - 6.1.1 API Documentation Improvement . . . 24
  - 6.1.2 Separation of Development and Production Requirements . . . 24
- 6.2 Backend and Frontend Containerization with Docker . . . 25
- 6.3 Docker Compose Implementation . . . 26
- 6.4 Building Workflows with GitHub Actions . . . 27
- 6.5 Carbon Emissions Tracking with CodeCarbon . . . 28

**7 Monitoring** **31**
- 7.1 Corrections . . . 31
- 7.2 Drift Analysis with Alibi Detect . . . 32
- 7.3 Drift Detection with Deepchecks . . . 33
- 7.4 Performance Monitoring with Prometheus . . . 34
- 7.5 Data Visualization with Grafana . . . 36
- 7.6 Cloud Deployment on Okteto . . . 37
- 7.7 Monitoring with Better Uptime . . . 38
- 7.8 Load Testing with Locust . . . 38
- 7.9 Corrections . . . 43

**8 Conclusion** **43**

# 1 Introduction

This report details the project developed by our team for the 2023-2024 "Software Engineering for AI-Enabled Systems" course, focusing on integrating a pre-existing computer vision model used for crop segmentation in precision agriculture with modern software engineering methodologies and tools. The phases of the project detailed in this report include Inception, where foundational elements like repository creation and model development were established; Reproducibility, focusing on ensuring consistent and reliable results; Quality Assurance, ensuring the integrity and effectiveness of the code and the machine learning model; API development, which involved creating a robust and efficient interface; Deployment, which encompassed containerization with Docker, workflow building with Github Actions, and cloud deployment on Okteto; and Monitoring, which involved uptime monitoring, load testing, metrics visualization, and drift analysis. Each phase contributed significantly to the project's progression and offered valuable insights into the challenges and intricacies of engineering AI-based systems. Through this structured approach, the project aimed to demonstrate the effective integration of AI in a real-world application, emphasizing reliability, efficiency, and scalability.

# 2 Inception

The initial phase of the project involved establishing a code repository on GitHub and implementing the Cookiecutter Data Science structure, a logical, reasonably standardized, but flexible project structure for doing and sharing data science work.

## 2.1 Repository Creation

The GitHub repository was created to serve as the central hub for all project-related code and documentation. Cookiecutter Data Science, a framework for structuring data science projects, was adopted to organize the repository. This framework provides a comprehensive directory structure, which helps in maintaining a clean and manageable codebase. It includes predefined folders for datasets, notebooks, references, and scripts, ensuring that all project components are well-organized and easily accessible.

## 2.2 Model and Dataset Card Development

As part of the initial phase, the Model Card and the Dataset Card were created, in order to provide a concise and well-structured overview of the model and datasets used in the project.

- **Model Card:** The Model Card includes details about the model's developer, version, type, and training algorithms. It also outlines its primary intended uses and users, ethical considerations, and performance metrics. The model, a U-NET neural network, aims to perform semantic segmentation tasks, particularly in precision agriculture. Key performance metrics include pixel accuracy and intersection over union (IoU).
- **Dataset Card:** The Dataset Card describes the datasets used for training and evaluation. The training dataset, 'UAV Sugar Beets 2015-16', consists of aerial images of sugar beet fields, while the evaluation dataset, 'Remote Sensing 2018 Weed Map', includes labeled aerial images for semantic weed mapping.

The card details dataset structure, instances, and additional information relevant to understanding the data's origin and composition.

# 3 Reproducibility

## 3.1 Corrections

After the feedback and instructions received during the presentation of the first milestone, the following changes were made:

- metadata was added to the model card
- a link from the model card to the dataset card was added
- a link from the README of the repository to the model card was added
- the model card was moved into the model folder and renamed README
- the dataset card was moved into the data folder and renamed README and the .gitignore file was modified accordingly

## 3.2 Data Versioning with DVC

This project milestone began with the initialization of DVC, which created the necessary configurations for data versioning. A remote storage on Google Drive was then configured to store DVC-tracked files, providing shared access and backup of data. The raw data was added to DVC, with adjustments made to the .gitignore file to handle the data directory properly and prevent accidental upload to GitHub.

When the raw data was updated, a change in the hash value in DVC files indicated a new version of the data. A global configuration file was created to manage settings for data paths, model training parameters, and other project-wide settings. The project's README file and dependencies were updated to reflect the current project structure and ensure compatibility and reproducibility.

Scripts for data extraction from a .zip file and utility functions necessary for the project were uploaded, and the script that prepares the dataset for training and testing was updated. The script defining the machine learning model was uploaded, and issues in train_model.py, predict_model.py, and make_dataset.py were fixed.

Processed data and the trained model were added to DVC for versioning, and a mechanism to report training and validation metrics was added. A DVC pipeline was created to automate and version the data preparation, training, and evaluation process.

Due to the blocking of the Google Drive account owing to excessive traffic, modifications were made to the dvc.yaml file to reduce the redundancy of files and to only version essential files, minimizing the occupied space. A notable change included using *persist: true* in the DVC configuration for the trained model file. This change allows the execution of the pipeline without the need to retrain the model, significantly reducing the time required to run the pipeline.

### 3.3 MLflow Integration

The integration of MLflow into the evaluation script marked the beginning of this phase, allowing for the logging of metrics, parameters, and artifacts to facilitate experiment tracking. To ensure reproducibility and ease of setup for experiment tracking, MLflow was added to the project's dependencies.

Four different experiments were then conducted by varying the model's hyperparameters to perform experiment tracking with MLflow. These experiments were conducted with the following configurations:

- Experiment 1: learning rate 0.001, batch size 50
- Experiment 2: learning rate 0.1, batch size 25
- Experiment 3: learning rate 0.001, batch size 25
- Experiment 4: learning rate 0.1, batch size 50

MLflow was used to ensure systematic logging and tracking of different configurations and their corresponding outcomes, providing a structured approach to experiment management.

The dvc.lock and dvc.yaml files were updated to reflect the different experiments that were conducted. The dvc.lock file was updated with the outputs of different experiments, including model files and metrics. In dvc.yaml, the output files were updated to include the persist: true flag, ensuring the preservation of model files across different pipeline runs. These updates ensure a smooth interaction between DVC and MLflow, where DVC manages the data and model versions while MLflow handles experiment tracking.

The src/config.py file was updated to include a dictionary of hyperparameters for grid search, and the src/models/train_model.py file was modified to implement a loop over the hyperparameter grid. This setup facilitated the systematic exploration of hyperparameter space and the logging of different experiment configurations and outcomes with MLflow.

The dvc.yaml file was updated to include the metrics files generated during the model evaluation phase, allowing DVC to track and version the metrics files and enabling a comprehensive examination of model performance across different hyperparameter configurations. The src/models/predict_model.py was modified to update the metrics logging mechanism to ensure structured logging of evaluation metrics, which is essential for assessing and comparing model performance.

The project's folder structure was updated, with the README file reflecting the structure of the project. An mlruns directory was added for MLflow experiment tracking data, and within the src directory, the model architecture was separated from the training and prediction script, improving modularity and clarity. Finally, the pipeline was executed and the generated mlruns folder, containing the logs and artifacts from the experiments, was uploaded to the repository.

### 3.4 DagsHub

A repository was first created within the se4ai2324-uniba organization on DagsHub, which was then connected to the GitHub repository of the project. Following this, the tracking of experiments conducted in the project and logged with MLFlow was enabled for DagsHub, integrating experiment tracking into the project workflow.

# 4 Quality Assurance

After the feedback and instructions received during the presentation of the previous milestone, the best model was added to the MLflow model registry.

## 4.1 Quality Assurance with Pytest

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

The model training testing ensured that the model trains successfully and is able to run on various devices. The tests developed for this purpose are detailed below.

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.

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.

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.

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.

In order 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.

Following this, behavioral testing was implemented to evaluate the model's behavior and response to different inputs and conditions. The developed tests are described below.

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. **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.

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.

In order 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.

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.

The screenshots in Figure 1 showcase the successful execution of all the implemented tests.

## 4.2 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 Figure 2.

- **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.

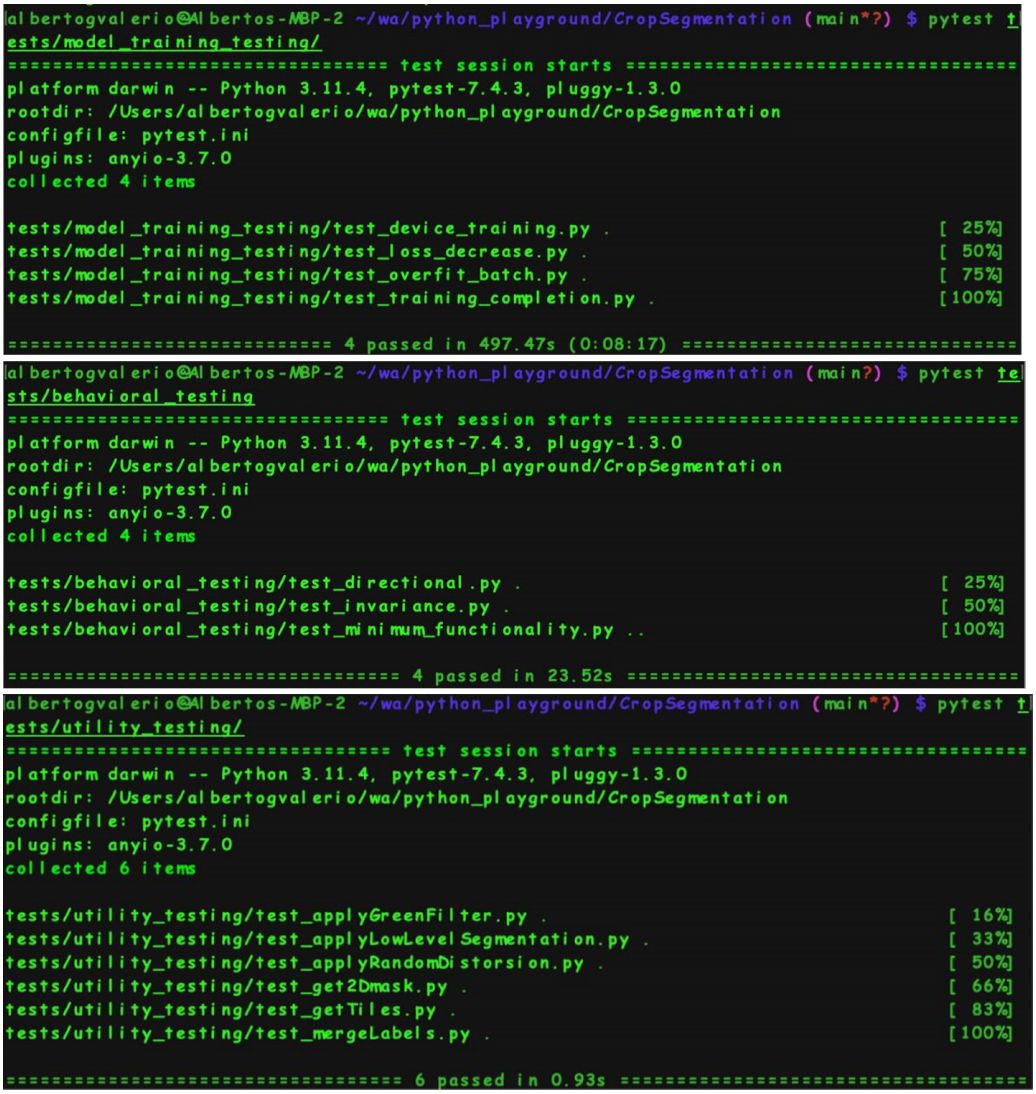


Figure 1: A screenshot showcasing the successful execution of the developed tests.

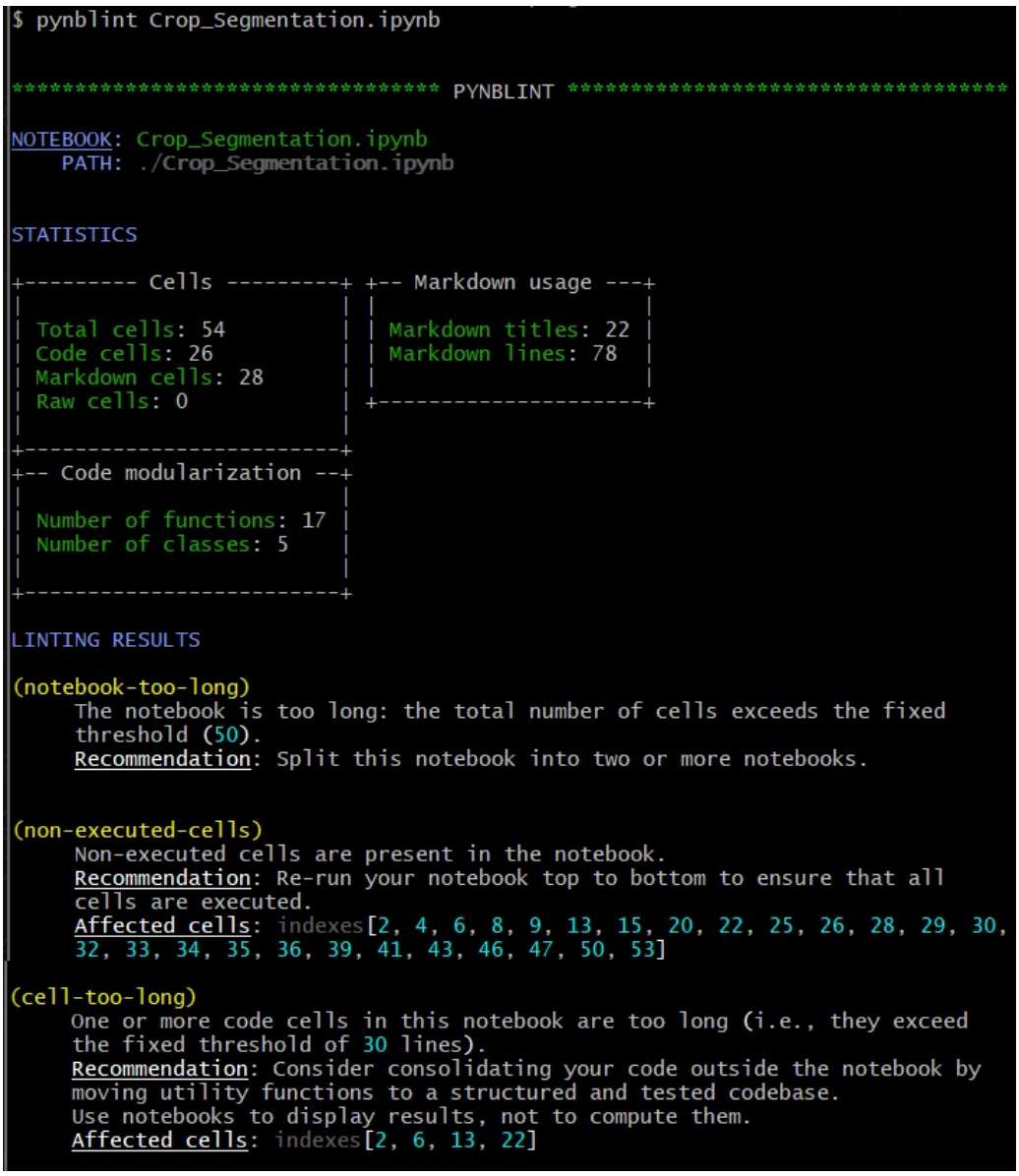


Figure 2: A screenshot showcasing initial Pynblint output.

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 Figure 3.

- **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.

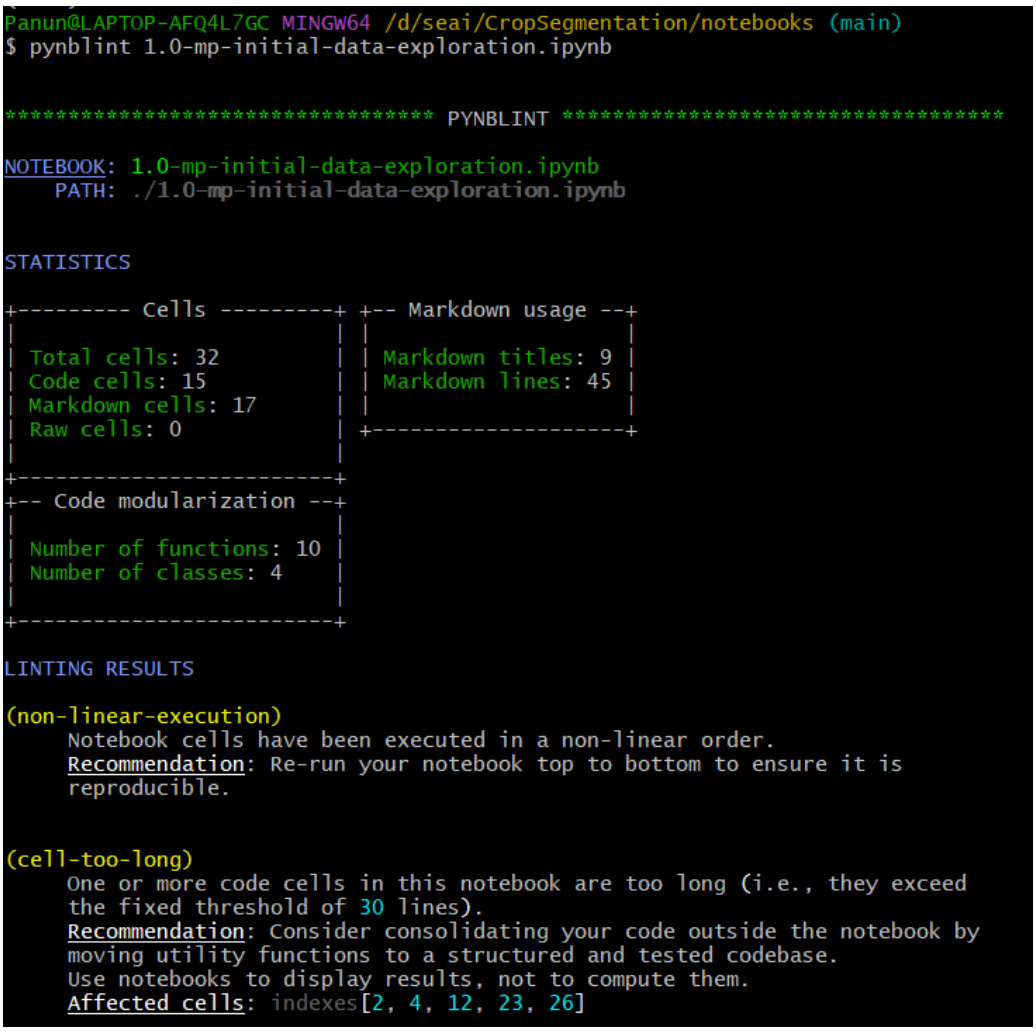


Figure 3: A screenshot showcasing Pynblint output after corrections.

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 Figure 4:

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


Figure 4: A screenshot showcasing final Pynblint output.

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.

## 4.3 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. Figure 5 showcases a representation illustrating the distribution of the detected issues among the different categories.

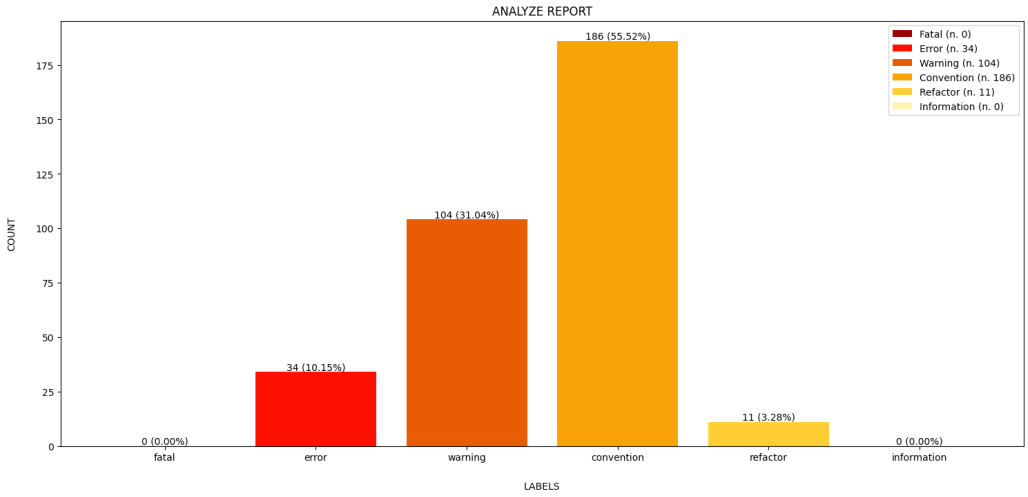


Figure 5: A screenshot showcasing initial Pylint output.

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 Figure 6, showcasing the significant improvement in code quality.

## 4.4 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 Figure 7.

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 represented in Figure 8.


Figure 6: A screenshot showcasing final Pylint output.

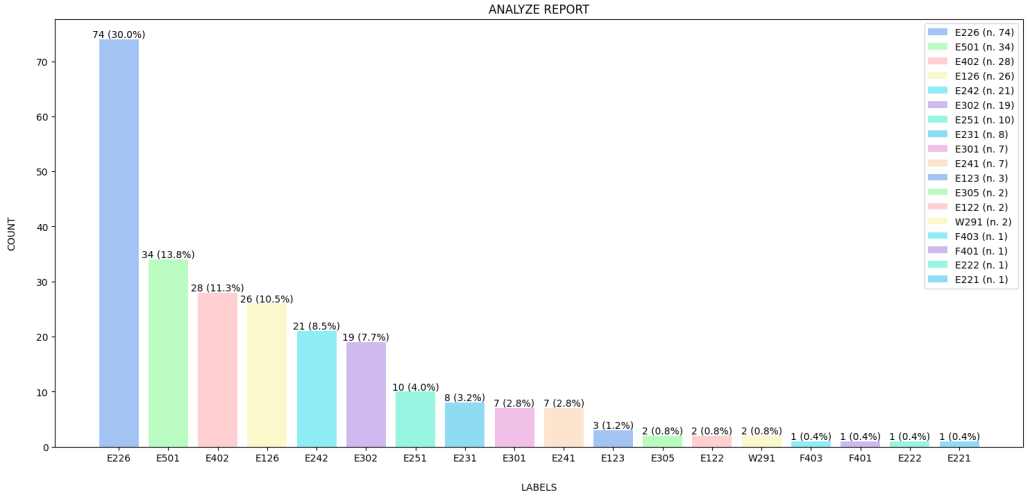


Figure 7: A screenshot showcasing initial Flake8 output.

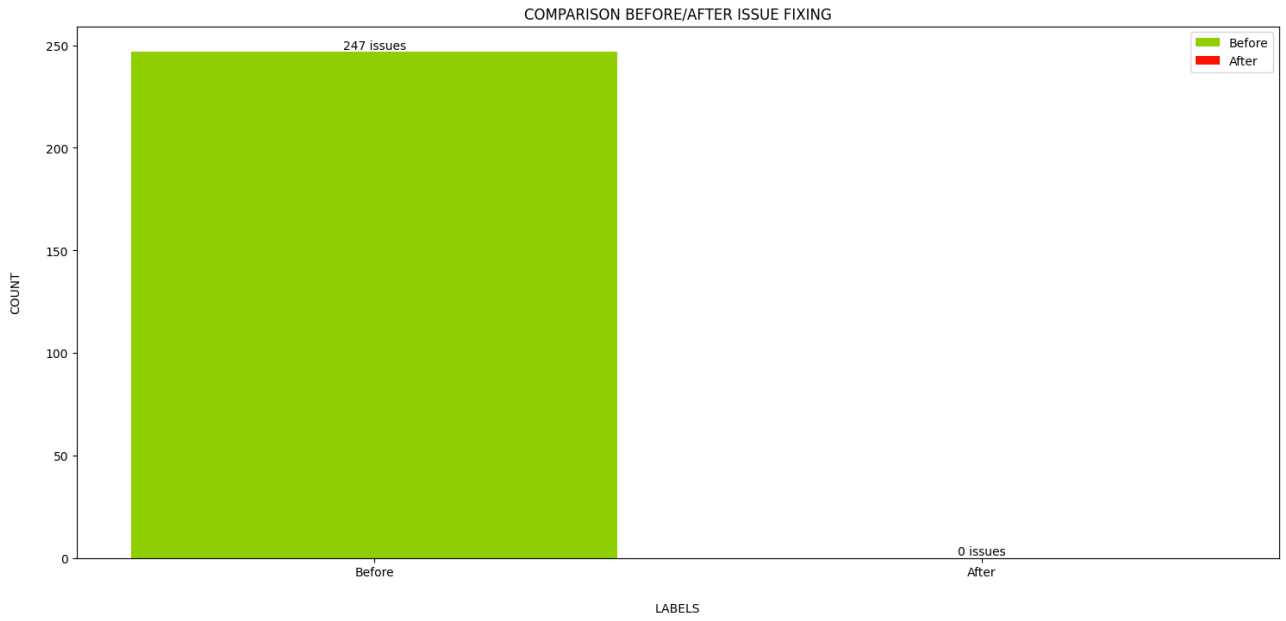


Figure 8: A screenshot showcasing final Flake8 output.

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.

## 4.5 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. 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. 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. 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.

### 4.5.1 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. An analysis for Data Integrity, Validation, and Model Evaluation was performed and the output files were saved in the project repository.

Below details are provided for the properties that are applicable for our project and that were verified by

Deepchecks.

**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**

- **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)].

#### 4.5.2 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.

# 5 API

## 5.1 Corrections

After the feedback and instructions received during the presentation of the previous milestone, the presentation documents for the previous milestones were added to the repository, as part of the artifacts of the project.

## 5.2 API Endpoint Implementation with FastAPI

FastAPI, a modern web framework for building APIs with Python, was used in our project to implement a suite of endpoints that enhance the interaction with our computer vision model and the handling of image data for segmentation tasks. A detailed description of each endpoint is provided below.

- **Main (GET):** Serves as a test endpoint, providing general information about the API and the system, including a dynamic list of requirements, ensuring that all dependencies are transparent and traceable. The functionality of this endpoint is showcased in Figure 9.

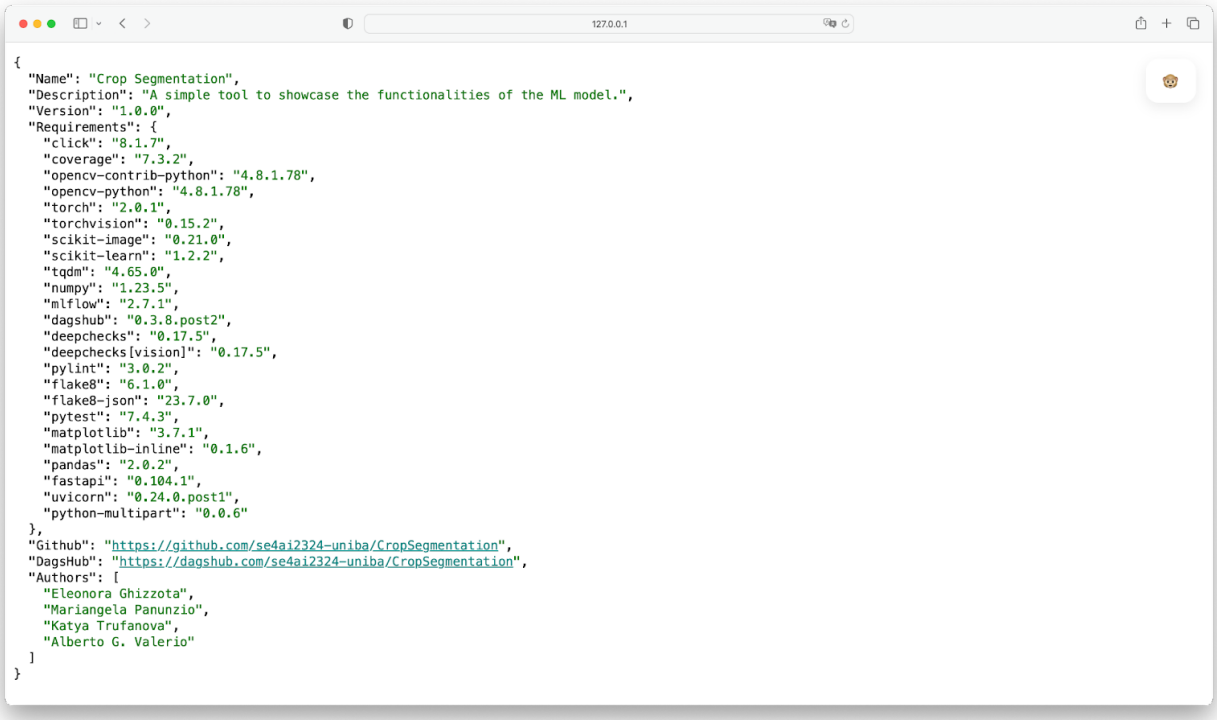


Figure 9: A screenshot showcasing the functionality of the Main (GET) endpoint.

- **Samples (GET):** Returns a random selection of images from the testing set, excluding any that are entirely black. This selection is stored in a temporary cache folder, and their filenames are returned in a JSON format by the server. An optional query string parameter, 'limit,' controls the sample size. In the absence of this parameter, the default behavior is to return the entire set.

- **Image (GET):** This endpoint simplifies image retrieval, providing a parameterized URL that returns a valid image stored in the server's temporary cache folder.

- **Predict (POST):** Performs segmentation mask detection given an image passed mandatorily in the body of the request.

- **Upload (POST):** Allows users to upload an image from their local device memory to the server. The image is transmitted as a byte stream in the body of the request, adhering to common web standards.

- **Metrics (POST):** Calculates reference metrics used throughout the project development process (accuracy and Jaccard index) from a predicted segmentation mask passed as a parameter, for which a ground truth exists in the testing set.

Throughout the development of these endpoints, we have ensured robustness and reliability. Each endpoint, in addition to parameter validation performed by Pydantic, is equipped to handle critical scenarios that may arise during method execution. Custom error messages are generated for various exceptions to inform the user of the specific issue encountered.

## 5.3 Ensuring API Robustness with Pydantic

In our project, Pydantic, an integral part of FastAPI, played a central role in improving the robustness and functionality of our API endpoints. Pydantic is a data validation and settings management library in Python, which is particularly beneficial for parsing and validating HTTP request bodies and query parameters. The primary function of Pydantic in our application was to validate the parameters for each API endpoint. This validation was crucial to ensure that the data received and processed by our endpoints was of the correct type and adhered to the defined constraints. Pydantic's validation mechanisms were used across all endpoints, including both mandatory and optional parameters. Where necessary, default values were provided to maintain the API's robustness and usability. Namely, Pydantic was implemented by creating the Image class. This class is a global model that includes two essential attributes: og_name and mask_name. These attributes represent the names of the original image and its corresponding segmentation mask, respectively. The Image class was primarily used in the 'Predict' and 'Metrics' routes of our API.

- In the 'Predict' endpoint, the og_name attribute of the Image class was used to reference the original image for which the segmentation mask was to be predicted.
- In the 'Metrics' endpoint, the mask_name attribute facilitated the retrieval of the predicted segmentation mask for the computation of key metrics like accuracy and the Jaccard index.

The integration of Pydantic into our FastAPI-based project provided several advantages:

1. **Type Safety:** Pydantic's robust type-hinting feature ensured that all data received by our endpoints was correctly typed, reducing the possibility of type-related errors.
2. **Error Handling:** Pydantic's built-in error handling mechanisms allowed for more informative and user-friendly error messages, enhancing the overall user experience.
3. **Simplicity and Efficiency:** The declarative nature of Pydantic models made the process of defining data structures and validation rules more straightforward, resulting in cleaner and more maintainable code.

## 5.4 API Documentation with ReDoc

FastAPI provides two different tools for API documentation: Swagger UI and ReDoc. Each of these tools offers unique features that are aimed at different needs and preferences. Swagger UI, served at the /docs endpoint by default, provides an interactive interface for testing API endpoints, which allows users to try out the API directly from the documentation, making it a practical choice for developers who want to test endpoints in real-time.

On the other hand, ReDoc, served at the /redoc endpoint, offers a modern and clean interface with robust support for markdown. This allows for rich text editing in the documentation, enhancing readability and user experience. While ReDoc lacks the interactive "try it out" feature of Swagger UI, its superior aesthetics and user-friendly design make it a good choice for projects that prioritize presentation and readability of the API documentation.

In this project, we chose ReDoc over Swagger UI. This decision was motivated by the need for a modern, clean, and highly navigable interface that enhances the user experience. ReDoc's support for markdown allows for comprehensive and well-structured documentation. The absence of the "try it out" feature in ReDoc can also be seen as a security measure, preventing potential misuse of the API directly from the documentation. Given these motivations, we believe that ReDoc was the more suitable choice for this project, aligning more closely with our needs for superior user experience and user-friendly design.

A custom script was created to export the ReDoc documentation page into a standalone HTML file, which was saved in the 'references' folder. This script uses the OpenAPI specification generated by FastAPI's app.openapi() method to create a complete HTML document with embedded ReDoc functionality. This approach allows for the generation of offline, portable API documentation that retains all the benefits of ReDoc's user-friendly interface and markdown support.

The documentation of our API, created using ReDoc, presents several detailed and advanced features that have significantly contributed to the quality and usability of our documentation:

1. **Structured and Comprehensive Endpoint Descriptions:** The ReDoc API documentation offers a well-structured and comprehensive overview of each API endpoint. These descriptions are complete with operational IDs and detailed summaries, facilitating an in-depth understanding of the purpose and function of each component of the API. This structured presentation can help developers and users in navigating through the functionalities offered by our API.

2. **Detailed Parameter and Response Schemas:** Our ReDoc documentation details the parameters and response schemas associated with each endpoint. This includes information on the type, required status, and comprehensive descriptions of each parameter and response body. Such detailed schema documentation can be essential for developers to understand the data requirements and expected outputs, thus improving the clarity and usability of the API.

3. **Emphasis on Error Handling Documentation:** An important aspect of our API documentation is its focus on error handling. The ReDoc documentation provides extensive information on the potential error responses for each endpoint, including HTTP status codes and descriptions of validation errors. This feature is particularly beneficial for developers in understanding the robustness of the API and can help in effective client-side debugging.

4. **Offline Accessibility:** A significant advantage of our documentation approach is the creation of a standalone HTML file, ensuring that the API documentation is accessible offline. This feature is particularly beneficial for environments with limited internet access and can serve as a reliable resource for archival and reference purposes.

## 5.5 Quality Assurance with Pytest for API Endpoints

In order to continue maintaining code quality and reliability in our project, Pytest was used to validate the implemented API endpoints for continued quality assurance. This section outlines the approach and specific tests implemented to ensure the robustness and efficiency of the API functionalities. The tests cover a wide range of API functionalities, from basic information retrieval to more complex operations like image upload.

This comprehensive coverage ensures that all critical aspects of the API are rigorously validated.

Below is a detailed overview of the implemented tests.

- **test_get_main:** This test ensures that general information about the API and the system, including a dynamic list of requirements, is correctly returned. The test verifies the proper functioning of the main endpoint, which is essential for providing users with the most important API details.

- **test_get_samples:** This test validates the functionality of the 'samples' endpoint, which is designed to return a selection of images based on the 'limit' parameter set by the user. For instance, setting limit=0 is expected to return the entire set of 102 images. This test is essential for confirming the endpoint's capability to handle different user-defined parameters and return the correct number of images.

- **test_get_image:** This test focuses on ensuring that only images from the set are returned and that invalid requests result in a 404 status code. This test is particularly important for validating the image retrieval functionality, ensuring it aligns with the expected naming convention and error handling.

- **test_post_predict:** This test uses parameterization to submit POST requests to the 'predict' endpoint with different image names. The process involves making a GET request to the '/images?limit=0' endpoint, followed by a POST request to the prediction endpoint with an image name in the JSON payload. The test asserts two key outcomes: a 200 status code response indicating a successful prediction request and the presence of a "mask" key in the JSON response, signifying the prediction of the segmentation mask. This approach is crucial for validating the accuracy and reliability of the image prediction mechanism.

- **test_post_metrics:** This test assesses the performance metrics of the model by submitting POST requests to the 'metrics' endpoint with different mask images. It verifies the server's response (200 status code) and the presence of essential metrics in the JSON response, including accuracy and intersection over union. This test is useful in providing insights into the model's accuracy and segmentation quality.

- **test_post_upload:** This test validates the 'upload/image' endpoint by simulating the uploading of an image file. It involves creating a synthetic RGB image, serializing it, and sending it as a file to the server via a POST request. The test asserts a successful upload response (200 status code) and the correct JSON response. This test is key in verifying the core functionality of the image upload endpoint.

The screenshot in figure 10 showcases the successful execution of all the implemented tests.

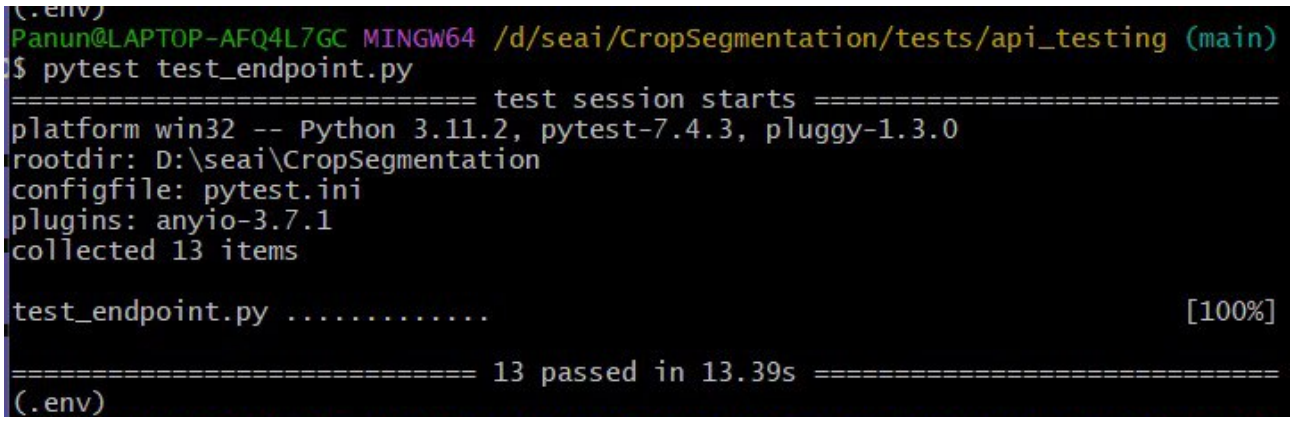


Figure 10: A screenshot showcasing the successful execution of the API endpoint tests.

### 5.6 Showcasing API Capabilities through an Interactive Web Application

In addition to the development of the API for our project, we also developed an interactive web application that serves as a front-end interface, designed to demonstrate the primary functionalities of the associated API. The application structure includes three key components: an HTML file for layout, a CSS file for styling, and a JavaScript file for interactive features. The web application is structured to facilitate three main functions: image selection, segmentation mask prediction, and calculation and display of performance metrics.

- **Image Selection:** The application allows users to select images in two ways: either from a pre-selected set extracted from the testing set (excluding entirely black images) or by uploading an image from their device. This second option ensures meaningful results only if the uploaded images align with the type of images used in the training set. The selection process is designed to be straightforward, ensuring ease of use.

- **Mask prediction:** Once an image is selected, the system performs a segmentation mask prediction. This function illustrates the practical application of our model by generating and displaying the segmentation mask directly on the user interface. It serves as an immediate visual representation of our model's capabilities in real-time applications. This functionality is showcased in Figure 11.

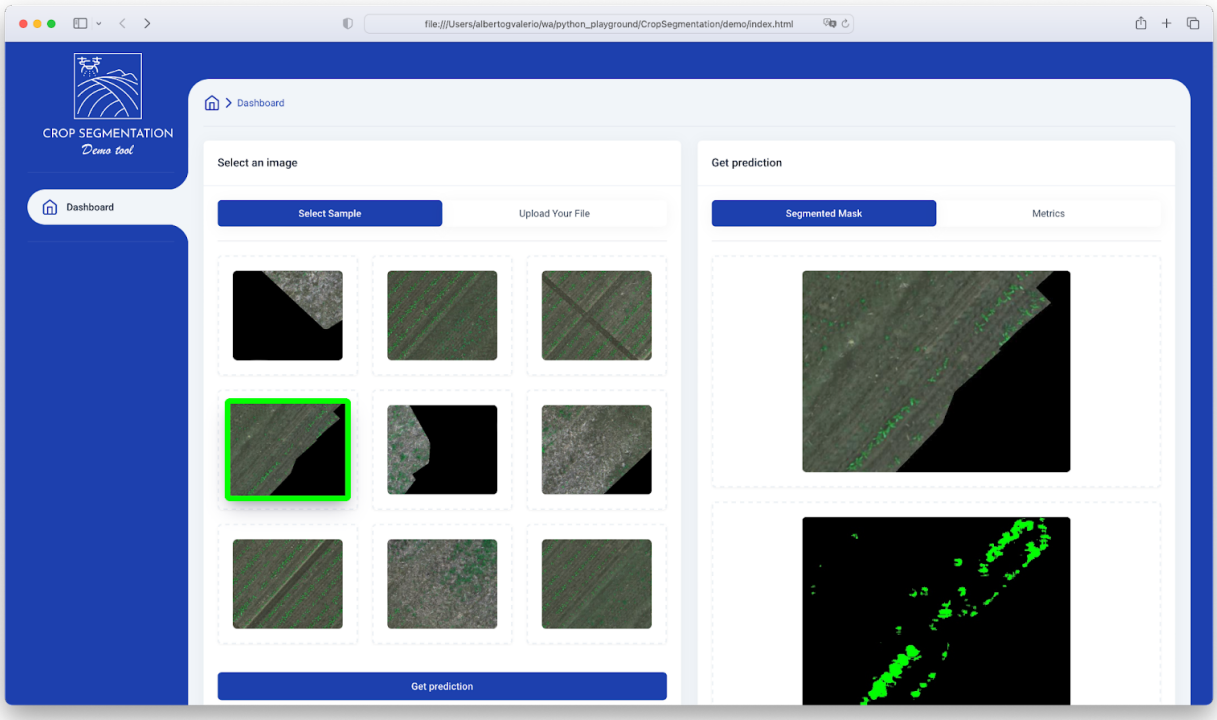


Figure 11: A screenshot showcasing mask prediction functionality.

- **Metrics:** Another significant functionality of our web application is the calculation and display of key metrics. This section becomes active only if the prediction is performed on an image chosen from the testing set, as this allows for the comparison with a ground truth. The metrics displayed include accuracy and the Jaccard index, both essential in evaluating the model's performance. The application not only presents these metrics but also contextualizes them by comparing them to the average values calculated across the entire testing set. This comparison is visually represented through a unique system of indicators: a double horizontal arrow indicates that the metric for the selected image falls within a [-10%, +10%] range of the average values, while upward and downward arrows denote metrics falling outside these bounds on the higher or lower end, respectively. This approach in metric presentation provides users with a clear understanding of how a particular image's analysis compares to the overall model performance. This functionality is showcased in Figure 12.

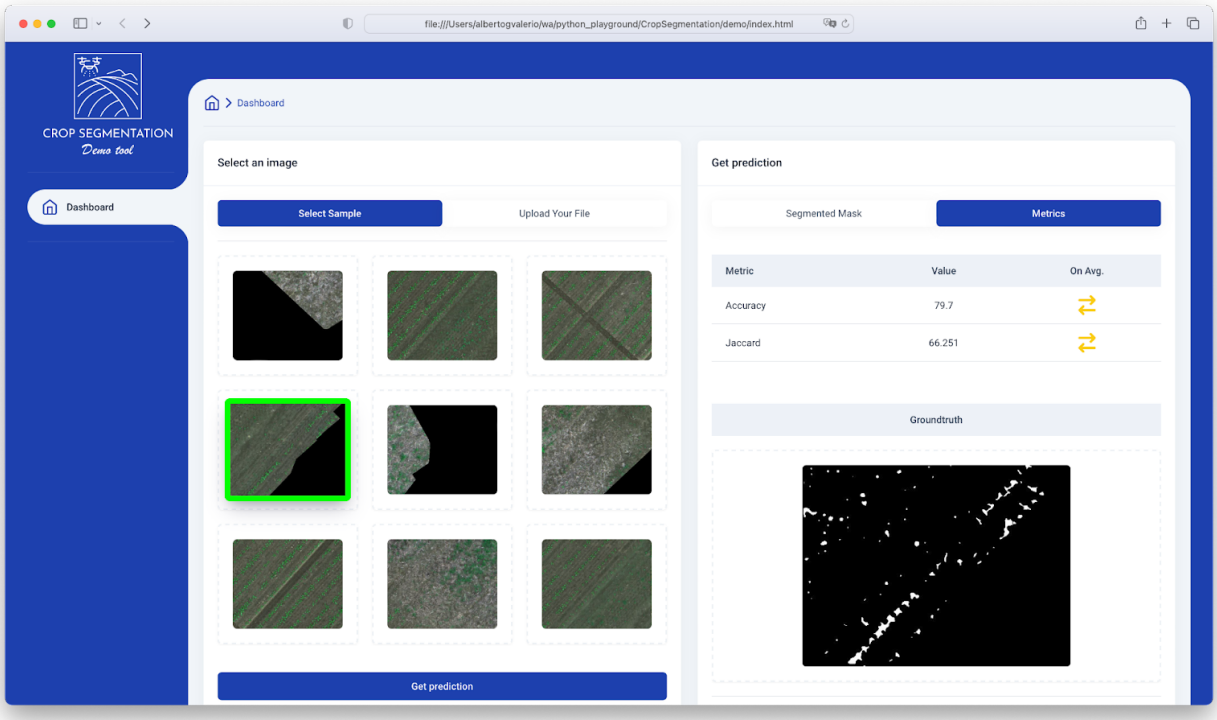


Figure 12: A screenshot showcasing the display of metrics.

# 6 Deployment

## 6.1 Corrections

### 6.1.1 API Documentation Improvement

To enhance the readability and maintainability of the code, as well as to improve the API documentation, comments have been added to each FastAPI endpoint. These comments provide a comprehensive description of the endpoint, its inputs, and outputs. This practice is crucial for facilitating the understanding of our codebase by providing detailed descriptions of parameters and returns in API documentation. This can help bridge the gap between the API creators and API users by providing clear and concise information about the API's capabilities, inputs, outputs, and error handling. Additionally, this improved API documentation can aid in code maintenance, since it makes it easier to update or modify the API as it evolves. After these changes, we re-ran the script to create the HTML file containing the ReDoc API documentation, ensuring that the documentation is up-to-date and accurately reflects the current state of the API.

### 6.1.2 Separation of Development and Production Requirements

Additionally, the requirements for the project were separated into development requirements and production requirements, in order to abide by the best practices for requirement definitions in software development. This approach allows for a clear distinction between the dependencies needed for development and those needed for the production environment. It also helps to manage the project's dependencies more efficiently and reduces the risk

of unnecessary dependencies in the production environment. The requirements-dev.txt file includes additional tools necessary for development, such as coverage for code coverage analysis, pylint and flake8 for code linting, and pytest for running tests. The deepchecks package is also included for data integrity checks. On the other hand, the requirements-prod.txt file lists the packages necessary for the application to run in the production environment. These include packages for data processing (numpy, pandas, scikit-learn, torch, opencv-python), web application (fastapi, uvicorn, python-multipart), and data visualization (matplotlib, matplotlib-inline). By separating the requirements, it becomes easier to manage the project's dependencies and ensure that only necessary packages are included in the production environment. This approach also makes it easier to update or modify dependencies for development or production independently. An important aspect to note is that the requirements-dev.txt file imports the requirements-prod.txt file. This means that instead of rewriting the same requirements, the development requirements file reuses the production requirements. This practice is beneficial for several reasons:

- **Clean Code:** It avoids code duplication, which is a fundamental principle of clean code. Duplication can lead to errors and inconsistencies, especially when updates are needed.
- **Modularity:** It enhances modularity as each file has a specific purpose and can be updated independently.
- **Maintainability:** It improves maintainability as changes in the production requirements automatically reflect in the development requirements.
- **Efficiency:** It increases efficiency as there's no need to manually synchronize the two files.

This approach of reusing code aligns with the DRY (Don't Repeat Yourself) principle, a best practice in software development. It contributes to a more manageable, efficient, and error-free project.

## 6.2 Backend and Frontend Containerization with Docker

Docker is a platform that enables developers to package and distribute applications in a lightweight, portable manner. This process, known as containerization, encapsulates an application with its environment and dependencies into a container. This ensures that the application works uniformly across different environments. Docker Compose further extends Docker's capabilities by allowing the definition and orchestration of multi-container Docker applications.

During the implementation of Docker and Docker Compose in our project, we followed established best practices in order to utilize containerization benefits to the fullest. For our project, two different Dockerfiles were implemented in order to create two separate containers for the backend and frontend with the goal of creating a scalable, maintainable, and efficient development and deployment workflow.

Below is a detailed description of the backed Dockerfile:

1. **Python Environment Management and Version Control:** The Dockerfile begins by specifying a Python slim base image, an efficient choice due to its smaller size. The Python version is defined as an argument (ARG PYTHON_VERSION=3.11.4), providing flexibility in upgrading or downgrading Python versions without altering the Dockerfile's main structure.

2. **Working Directory Set-Up:** Initially, the working directory is set to the root and later changed to /src/api. This facilitates a clear and organized structure within the Docker container and segregates the application code from other files, enhancing readability and maintainability.

3. **Environment Variables:** Two critical environment variables are set: PYTHONDONTWRITEBYTECODE to prevent Python from writing .pyc files and PYTHONUNBUFFERED to ensure real-time log outputs. This aids in debugging and monitoring.

4. **System Dependency Installation:** The Dockerfile installs necessary system dependencies (like gcc, ffmpeg, libsm6, and libxext6), indicative of the project's reliance on media processing and advanced computational functionalities.

5. **Optimized Python Dependency Handling:** The Dockerfile employs a layered approach to manage Python dependencies. By separating the downloading of dependencies from the installation, the build process leverages Docker's cache. This method significantly reduces build time, especially when there are no changes in the requirements.txt file.

6. **Code Copying Strategy:** It copies the entire source code into the container, as opposed to selectively adding files, simplifying the deployment process.

7. **Exposing and Running the Application:** The Dockerfile exposes port 80 and specifies the command to run the application using uvicorn, thus making the project's web application ready for production.

The frontend Dockerfile uses Nginx, a popular web server, for the containerization of the project's web-based frontend application. It copies the Nginx configuration and the web content into the container. Subsequently, it exposes port 8080, which aligns with standard web application practices and ensures the application is accessible. The dimensions of the backend and frontend Docker images are provided in Figure 13.

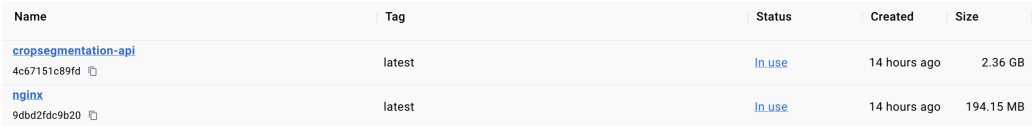


Figure 13: A screenshot showcasing Docker image sizes.

## 6.3 Docker Compose Implementation

The compose.yaml file for our project defines two services: the API and the frontend application. This setup demonstrates a microservices architecture, ensuring scalability and maintainability. The service configuration within the compose.yaml is concise yet comprehensive, detailing the build context and port mappings. This clarity facilitates easy updates and maintenance. The frontend Dockerfile maps the container ports to the host machine's ports (80 and 8080) ensuring accessibility of the services.

The Docker and Docker Compose implementation in this project follows established best practices:

- **.dockerignore:** The use of a .dockerignore file is a best practice that enhances build efficiency by excluding unnecessary files from the build context.

- **Layer Chaching:** The Dockerfiles demonstrate best practices in layer caching, a method crucial for optimizing build times and resource usage.
- **Security and Slim Images:** Using slim images and minimizing the number of layers reduces the security risk and the overall size of the images.

## 6.4 Building Workflows with GitHub Actions

GitHub Actions is a tool provided by GitHub that allows developers to automate, customize, and execute their software development workflows right in their repositories. Github Actions are custom automated sequences organized into workflows, where each workflow consists of one or more jobs that can run in sequence or in parallel. Within a job, a series of steps execute commands, scripts, or actions. GitHub Actions are event-driven, meaning they can be triggered by specified events such as a push or pull request. They are defined in YAML files that are saved in the .github/workflows folder of the project and can be used for a variety of tasks such as testing code, deploying software, and much more. Another important feature is represented by Artifacts, which are files that are produced by a step in a GitHub Actions workflow. They can be used to share data between jobs in a workflow and can be downloaded after a workflow run finishes.

In our project, we have utilized GitHub Actions to automate several aspects of our software development process. We have implemented workflows for testing our model, testing our API, and checking the quality of our code with Pylint, Flake8, and Pynblint. These workflows are triggered by specific changes in our repository.

The following workflows were implemented to automate and streamline our development process:

1. **Testing Model Workflow:**The testing_model.yaml workflow is triggered when changes are made to the src folder, excluding the src/api folder, or to any of the testing folders. This workflow sets up Python, installs the necessary dependencies, sets up the data, and then runs behavioral, model training, and utility tests using Pytest.
2. **Testing API Workflow:** The testing_api.yaml workflow is triggered when changes are made to the src/api folder or the tests/api_testing folder. This workflow also sets up Python, installs the necessary dependencies, sets up the data, and then runs API tests using Pytest. The integration of DVC data with GitHub Actions was accomplished by setting up DVC in the workflow environment and establishing a Google Drive Remote by using a service account for authentication, using the GDRIVE_CREDENTIALS_DATA secret. The workflow then retrieves the necessary datasets with dvc pull -r myremote and extracts the zipped datasets into the specified directory for further use in testing.
3. **Flake8 Code Quality Check Workflow:** The flake8_code_quality_check.yaml workflow is triggered when changes are made to the src folder. This workflow sets up Python, installs Flake8, and then checks the quality of the code in the src folder using Flake8. The results of the Flake8 check are saved in a JSON file, flake8_report.json, which is then uploaded as an artifact.
4. **Pylint Code Quality Check Workflow:** The pylint_code_quality_check.yaml workflow is triggered when changes are made to the src folder. This workflow sets up Python, installs Pylint, and then checks the quality of the code in the src folder using Pylint. The results of the Pylint check are saved in a JSON file, pylint_report.json, which is then uploaded as an artifact.

5. **Pynblint Notebook Quality Check Workflow:** The pynblint_code_quality_check.yaml workflow is triggered when changes are made to the project notebook. This workflow sets up Python, installs Pynblint, and then checks the quality of the project notebook using Pynblint. The results of the Pynblint check are saved in a JSON file, pynblint_report.json, which is then uploaded as an artifact.

In the Flake8, Pylint, and Pynblint workflows, code quality reports are generated and saved as artifacts - outputs of each GitHub action check. These artifacts can be downloaded after the execution of the GitHub action, providing a detailed analysis of the code quality and notebook standards. This feature is particularly beneficial for continuous integration, as it allows us to review the code quality check results after each workflow run and promptly address any issues highlighted in these reports.

## 6.5 Carbon Emissions Tracking with CodeCarbon

In recent years, the environmental impact of computing, particularly in data-intensive fields like machine learning, has garnered increasing attention. Carbon emission tracking is a critical aspect of sustainable software development, allowing developers to quantify and minimize the environmental footprint of their computing processes. CodeCarbon, a library designed for this purpose, offers a straightforward way to measure and analyze the carbon emissions associated with computational tasks.

In our project, CodeCarbon was integrated for tracking carbon emissions during the training phase, a period known for high computational demand and, consequently, potentially significant carbon emissions. The carbon emission tracking was set up to encompass each epoch of the model's training phase. The following points highlight the key aspects of this implementation:

1. **Tracking Setup:** CodeCarbon was configured to track emissions from the start to the end of each training epoch. This granularity allowed for a detailed analysis of emissions over the course of the model's development.

2. **Data Collection:** The emissions data collected during each epoch were saved in a CSV file. This method provided a straightforward way to aggregate and analyze the total emissions resulting from the training of the model.

3. **Output Information:** The output data from CodeCarbon was categorized into three types, each contributing to the overall calculation of CO2 emissions, as can be seen in Figure 14: Consumption Information, Geographical Information, and Hardware Information. Consumption Information is data on the energy consumed during the computational process. Geographical Information includes details about the geographic location of the computing resources, which is relevant due to variations in energy sources and their carbon footprints across different regions. Hardware Information pertains specifications of the hardware used, such as CPU, GPU, and RAM, as these components have different energy efficiencies and impact the overall carbon emissions.

4. **Optimizations for Realistic Results:** To ensure the accuracy and relevance of the results, several optimizations were implemented. Firstly, the library was set to track the specific process involved in model training, rather than the entire usage of the machine's hardware. Secondly, unique identifiers were

| | duration | emissions | cpu_power | gpu_power | ram_power | cpu_energy | gpu_energy | ram_energy |
|---|---|---|---|---|---|---|---|---|
| 44 | 216.815621 | 0.000206 | 0.1177 | 0.0612 | 0.279013 | 0.000115 | 0.000446 | 0.000044 |
| 24 | 215.101522 | 0.000214 | 0.0588 | 0.0191 | 0.255592 | 0.000056 | 0.000536 | 0.000037 |
| 68 | 215.810879 | 0.000226 | 0.1174 | 0.0631 | 0.259741 | 0.000092 | 0.000535 | 0.000036 |

| | country_name | country_iso_code | region | longitude | latitude |
|---|---|---|---|---|---|
| 37 | Italy | ITA | apulia | 16.8547 | 41.1122 |
| 36 | Italy | ITA | apulia | 16.8547 | 41.1122 |
| 0 | Italy | ITA | apulia | 16.8547 | 41.1122 |

| | os | cpu_count | cpu_model | gpu_count | gpu_model | ram_total_size |
|---|---|---|---|---|---|---|
| 10 | macOS-13.6.2-arm64-arm-64bit | 12 | Apple M2 Max | 1 | Apple M2 Max | 64.0 |
| 63 | macOS-13.6.2-arm64-arm-64bit | 12 | Apple M2 Max | 1 | Apple M2 Max | 64.0 |
| 21 | macOS-13.6.2-arm64-arm-64bit | 12 | Apple M2 Max | 1 | Apple M2 Max | 64.0 |

Figure 14: CodeCarbon output: Consumption Information, Geographical Information, and Hardware Information.

used for different training runs, especially those with varying hyper-parameter configurations, represented in Figure 15. This approach was critical during the experimental phase for comparing different model iterations.

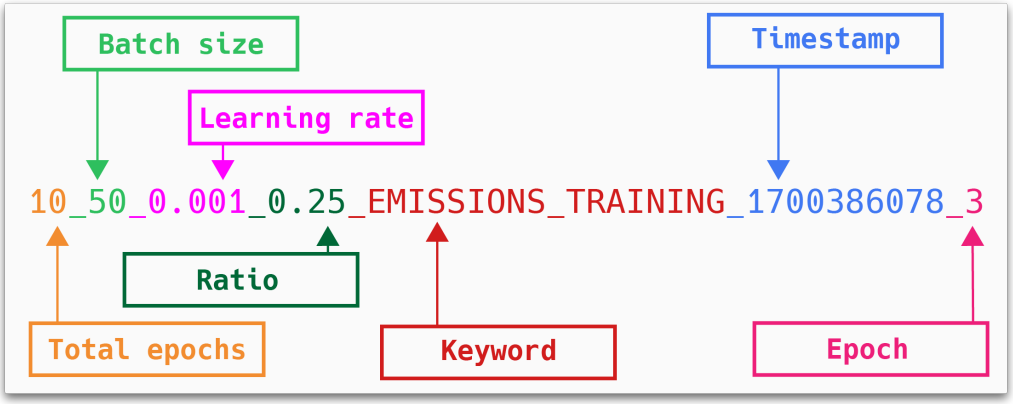


Figure 15: Representation of hyperparameter configurations.

5. **Trade-Off Analysis:** Further insights on the topic have been carried out, developed, and summarized in a Python notebook, where the experimental setting was used to search for the best hyper-parameters for the model based on grid-search, and implemented in the initial phase of the project. A key objective was to evaluate the trade-off between model performance (in terms of accuracy and Intersection Over Union - IOU) and carbon emissions. We found that models with lower performance metrics tended to consume more CO2, indicating a correlation between model efficiency and environmental impact, as shown in Figure 16

6. **Energy Consumption vs. Computing Power Analysis:** An additional aspect examined was the relationship between electrical energy consumption and the computing power of different machine components. It was observed that while the GPU was less energy-consuming, it provided significantly higher computing power, underscoring its importance in deep learning model training. Figure 17 showcases the average values obtained during the epochs of the training phase.

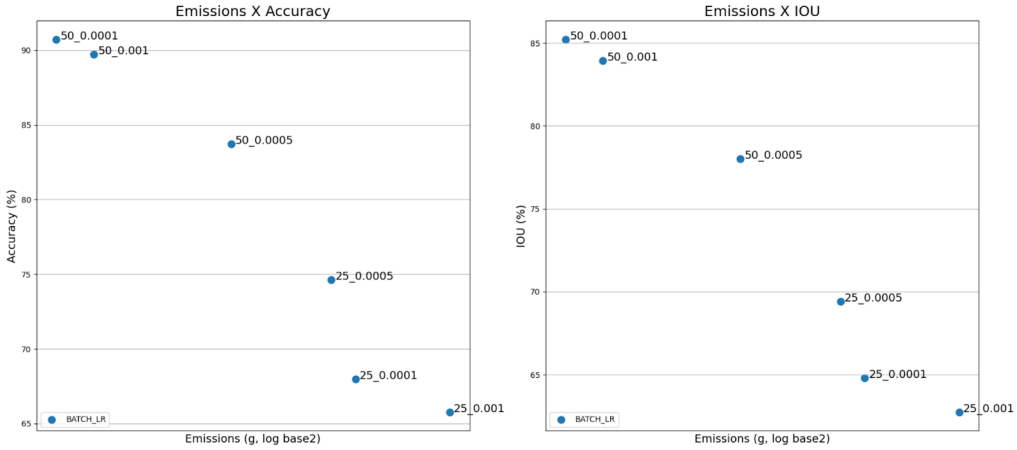


Figure 16: Representation of hyperparameter configurations.

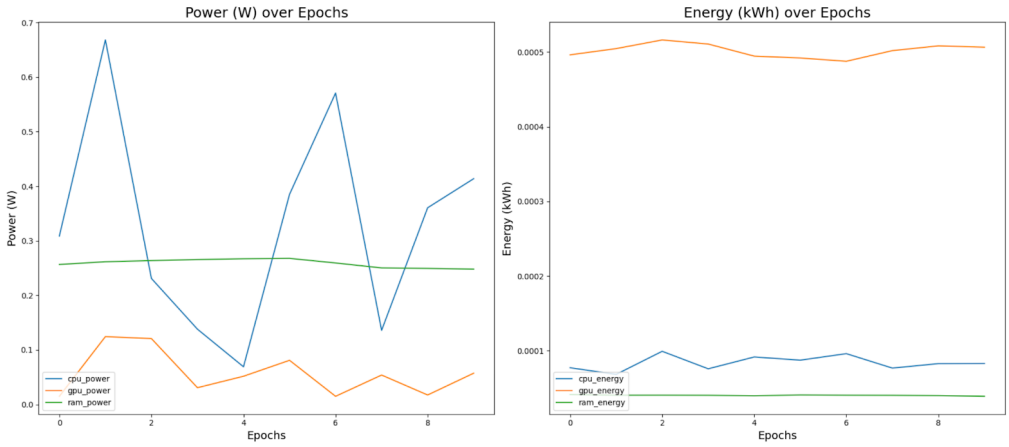


Figure 17: Energy Consumption vs. Computing Power Analysis.

7. **Energy Efficiency Classification:** The best-performing model was further analyzed using the Green AI Dashboard tool to obtain an energy efficiency classification. Our model achieved a Class B rating, as shown in Figure 18, reflecting a balance between computational efficiency and environmental responsibility.

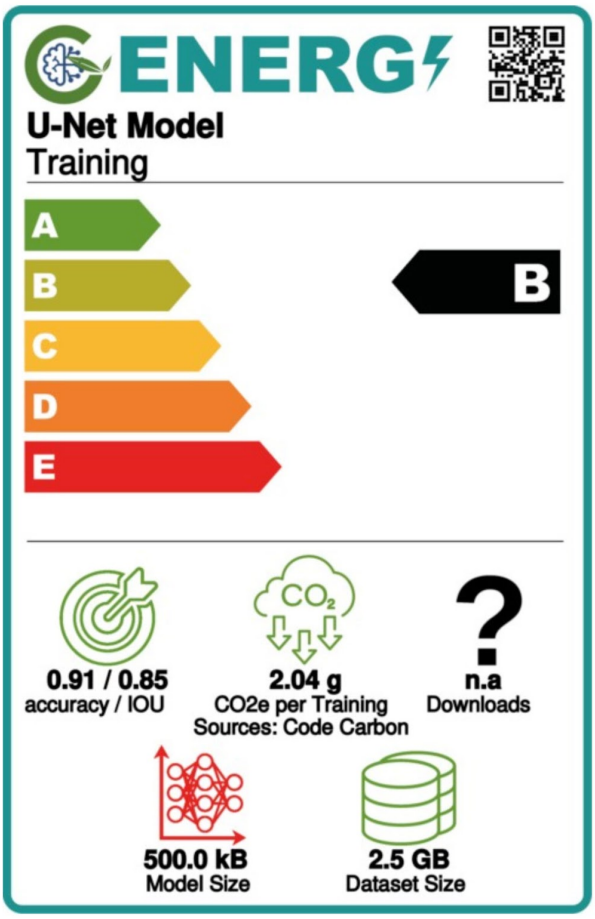


Figure 18: Energy Label for the best performing model.

8. **Model Card Update:** The project's model card was updated to include carbon emission metadata, a section describing the carbon footprint, and the energy efficiency label obtained from the Green AI Dashboard.

# 7 Monitoring

## 7.1 Corrections

As recommended during the review of the previous milestone, branch merging prevention was applied in the case of Github Action failure for all the workflows implemented during the previous milestone. This was done by using a branch protection rule for the main branch of the repository.

## 7.2 Drift Analysis with Alibi Detect

Alibi Detect is a Python library that can be used for drift detection, which involves identifying when the distribution of the input data changes over time, which can negatively impact the performance of a machine learning model. In our project, we utilized Alibi Detect to analyze data drift, specifically focusing on scenarios where a change in the detection sensor, sensor performance degradation, or changes in lighting conditions could lead to variations between training and testing data. We considered two specific cases: one where there was a reduction in brightness, and another where there was an increase in brightness.

We randomly selected a subset of images from our training data and converted them to grayscale. For each image, we computed a histogram representing the tonal distribution of the image. We then created two sets of artificially distorted images, one with reduced brightness and one with increased brightness, and computed their grayscale histograms. Figure 19 shows an original image from the training data, randomly selected, and compared with those where an artificial distortion of brightness has been applied. The histogram as a numerical feature of the image is considered, representing the tonal distribution of a digital image by plotting the number of pixels per tonal value. The horizontal axis of the graph represents tonal variations, while the vertical axis represents the number of pixels of that particular tone. Additionally, in Figure 19, we also present images showing brightness reduction and enhancement to visually represent these distortions alongside their respective histograms and cumulative distribution functions for a comprehensive analysis.

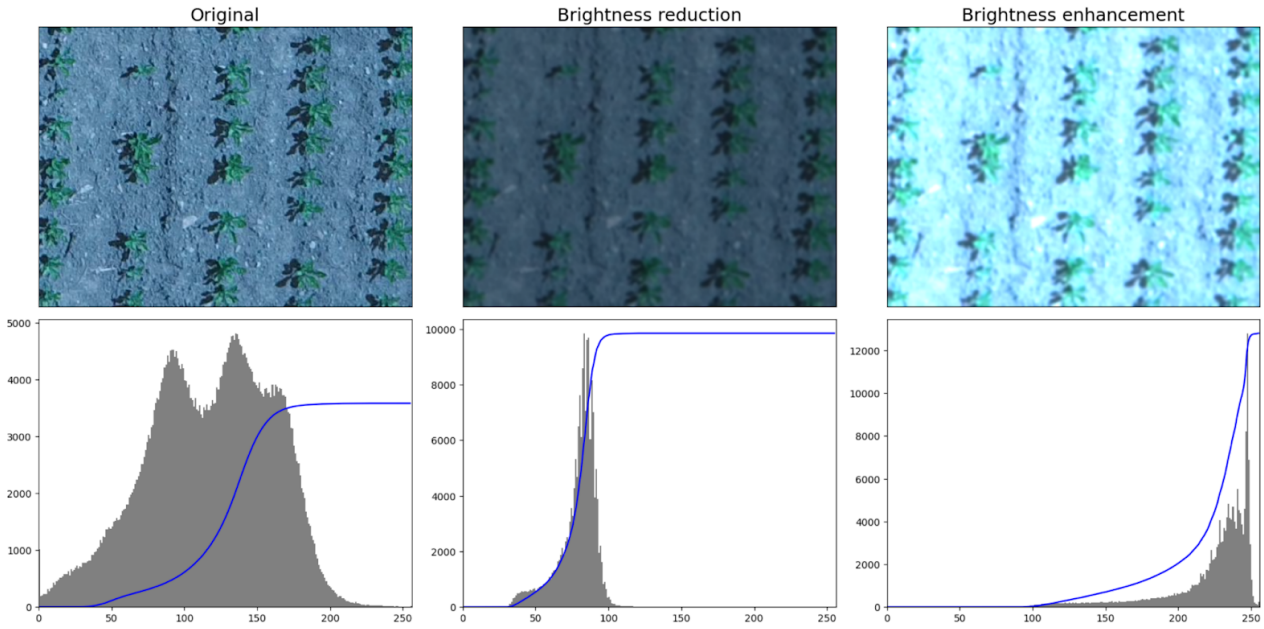


Figure 19: Original and corrupted images, with respective histograms.

It is evident that a reduction in light intensity shifts the tonal spectrum towards lower values (closer to zero), while an increase pushes the distribution towards higher values (closer to 255).

We then applied the Kolmogorov-Smirnov test to detect data drift, using the average histogram values obtained from the entire selected set composed of 7100 images. The distribution shown in Figure 20.

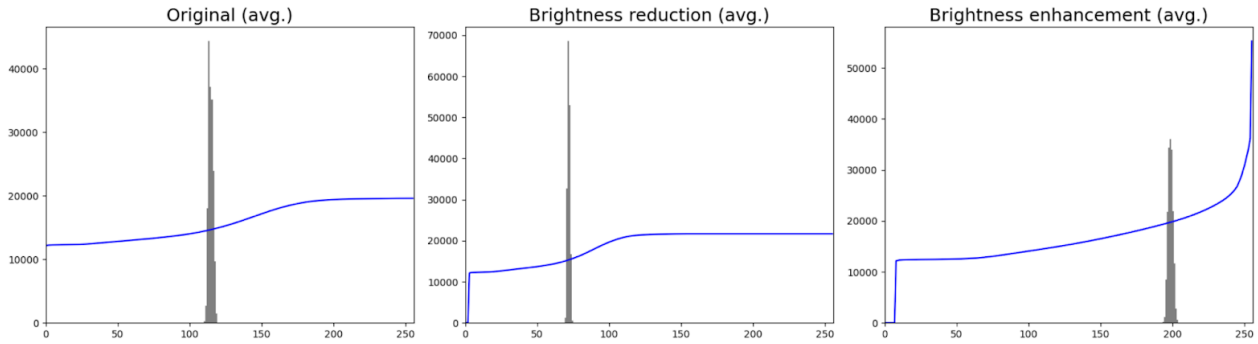


Figure 20: Data drift histograms.

The Kolmogorov-Smirnov test produced the values shown in Table 1 related to distance and p-value.

| **Property** | **Low images** | **High images** |
|---|---|---|
| Test name | KSDrift | KSDrift |
| Threshold | 0.01 | 0.01 |
| Drift | Yes | Yes |
| Distance | 0.3984375 | 0.3828125 |
| P value | 7.6135265e-19 | 2.2198778e-17 |

Table 1: Kolmogorov-Smirnov test results.

This analysis demonstrates the effectiveness of Alibi Detect in identifying and quantifying data drift in our project, providing valuable insights into the stability and reliability of our machine learning model under varying conditions. The visualizations and statistical tests allowed us to quantify the impact of brightness changes on the tonal distribution of our images, which is crucial for understanding how such changes could affect the performance of our model.

## 7.3 Drift Detection with Deepchecks

We used the checks provided by Deepchecks to measure the drift between training and test data. Namely, we measured how much the distribution of production data deviates from the training data and how this impacts the performance of the model. As done for Alibi Detect, a subset of images was altered by introducing Gaussian blur and randomly increasing or decreasing the brightness. Specifically, we performed the following checks on both the original data and the synthetic production data:

- **Prediction Drift:** This check detects prediction drift by using univariate measures on the prediction properties (samples per class, segment area, number of classes per image ). Prediction drift is when drift occurs in the prediction itself. Calculating prediction drift is especially useful in cases in which labels are not available for the test dataset, and so a drift in the predictions is a direct indication that a change that happened in the data has affected the model's predictions. For the calculation of the drift score, the Kolmogorov-Smirnov test and Cramer's V were used. The K–S test determines whether or not an empirical distribution conforms to a theoretical distribution, or if there is a significant difference between data distributions. The K-S test checks whether the null hypothesis (which is that the two samples come from the same distribution) is true. So, this metric calculates how much the distributions of two data

sets differ, together with a confidence score. If the confidence score is less than 0.05, the null hypothesis is rejected, indicating a model drift. The Cramer's V is a measure of association between two nominal variables, giving a value between [0,1], where 0 corresponds to no association between the variables and 1 to complete association. Table 2 showcases how the drift score increases up to 10 times on the production data.

| | Segment Area (K-S) | #Classes per Image (K-S) | Samples per Class (Cramer's V) |
|---|---|---|---|
| Original | 0.078 | 0.052 | 0.0 |
| Production | 0.468 | 0.56 | 0.165 |

Table 2: Drift Scores

- **Image Dataset Drift:** This check detects multivariate drift leveraging images properties (such as brightness, aspect ratio, RGB Relative Intensity). Image dataset drift is a drift that occurs in more than one image property at a time, and may even affect the relationships between those properties, which are undetectable by univariate drift methods. While no results are shown for the original data, implying that no drift was detected, the drift score reached 0.366 on the production data. This indicates a significant shift in the underlying data distribution between the training and production datasets. The features that contributed most to the drift are shown: in our case, the brightness is at the base of the differences between the training and production datasets. This suggests that the alterations made to the images, such as the introduction of Gaussian blur and random changes in brightness, have had a substantial impact on the model's ability to make accurate predictions. The model was trained on a specific data distribution, and these alterations have caused the production data to deviate from this original distribution.

- **Image Property Drift:** This check detects image property drift by using univariate measures on each image property (brightness, aspect ratio, RGB Relative Intensity, etc.) separately. In this case, image drift is a data drift that occurs in images in the dataset. Deepchecks shows us that, while no drift is detected in original data, when it comes to production data, two drifted properties stand out: brightness, as we expected, with a drift score of 0.387, and RMS contrast (contrast of the image calculated by standard deviation of pixels), with a drift score of 0.312. Other image properties remained stable. This outcome, as expected, predictably reflected the brightness alterations made to the production images.

In conclusion, the use of Deepchecks has allowed us to quantify and understand the impact of data drift on our model's performance. This understanding is vital for maintaining the accuracy and reliability of our model in a production environment. It highlights the importance of continuous monitoring and adjustment in response to changes in the data distribution. This is particularly relevant in dynamic environments where data can change over time.

## 7.4 Performance Monitoring with Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit and integrating it into our project allowed us to ensure the reliability and performance of our application; this tool provided insights into the performance and behavior of the application, allowing for thorough analysis. Below, an outline of the steps taken to integrate Prometheus monitoring into our project is provided.

1. Prometheus Configuration: The configuration file, prometheus.yml, serves as the starting point for establishing the monitoring framework. It sets the global scrape interval and external labels, laying the groundwork for the subsequent collection of metrics. The scrape_configs section defines the job for Prometheus, specifying the target as the FastAPI application running on localhost:5500. This configuration establishes the foundation for collecting metrics at regular intervals.

2. Monitoring Instrumentation: The monitoring.py file uses the prometheus_fastapi_instrumentator library to instrument the FastAPI application. The Instrumentator class is configured with options such as grouping status codes, ignoring untemplated paths, and instrumenting in-progress requests. Various metrics, including request and response sizes, latency, and total requests, are added using the metrics module. Additionally, a custom metric, segmentation_result, is introduced to track the outcome of image segmentation. This metric is associated with the /predict handler and records success and failure labels based on the X-image-segmentation-result header in the response.

3. Integration in server.py: The server.py file was updated to import the instrumentator from monitoring.py and incorporate it into the API. This was done by using the instrumentator.instrument(app) method, and the monitoring was excluded from the /metrics endpoint to prevent interference with the monitoring itself.

Figure 21 showcases Prometheus output.

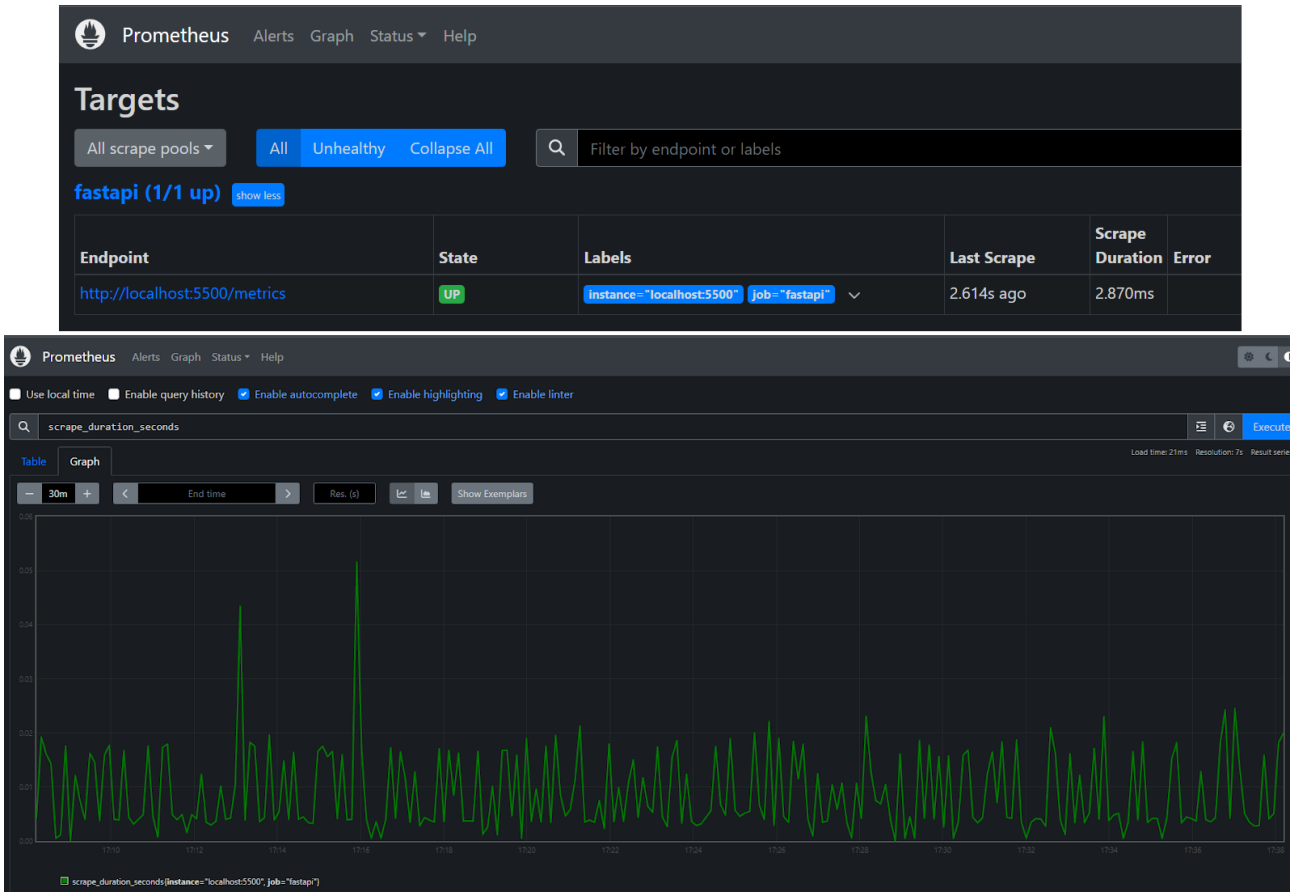


Figure 21: Screenshot showcasing Prometheus output.

## 7.5 Data Visualization with Grafana

Grafana is a multi-platform open-source analytics and interactive visualization web application. In our project, it was integrated with Prometheus in order to improve the visualization of different monitoring metrics implemented with Prometheus. Following the installation guidelines provided by Grafana, we successfully established access to the Grafana web interface. One of the key features of Grafana is its ability to directly integrate with data sources, and in this case, Prometheus is configured as the primary data source.

The URL and access settings were defined to establish a connection between Grafana and the Prometheus server to retrieve the metrics. All components, including our API (running at localhost:5500), Prometheus (running at localhost:9090), and Grafana (running at localhost:3000), were deployed locally. Following this, we created the dashboard by adding panels that define visualizations of specific metrics to it in order to provide a clear and informative representation of the data. Subsequently, Prometheus queries were executed to fetch relevant metrics. In order to enhance the coherence and aesthetics of the dashboard, we organized the panels into three rows.

Below, we detail the specific Prometheus queries used to fetch relevant metrics for our Grafana dashboard, organized into three categories: Python Garbage Collection Metrics, Scrape Metrics, and Image Segmentation Metrics.

Queries related to the Python garbage Collection Metrics row, showcased in Figure 22:

- python_gc_objects_collected_total: total number of objects collected by the Python Garbage Collector;
- python_gc_objects_uncollectable_total: total number of uncollectable objects identified by the Python Garbage Collector;
- python_gc_collections_total: total number of garbage collection collections performed by the Python Garbage Collector.

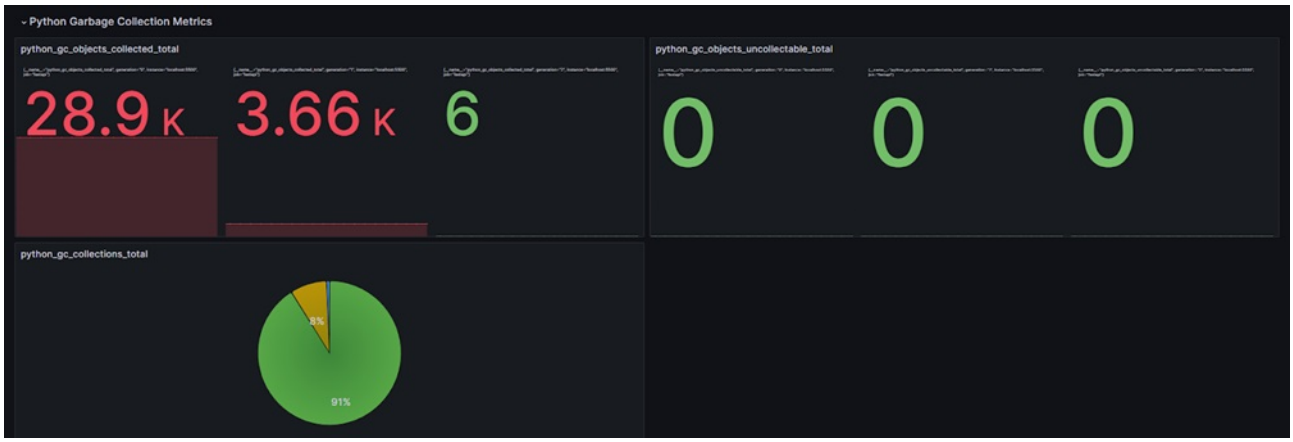


Figure 22: Garbage Collection Metrics.

Queries related to the Scrape Metrics row, showcased in Figure 23:

- scrape_duration_seconds: time spent by Prometheus to collect metrics from the target;

- scrape_samples_post_metric_relabeling: number of samples obtained after metric relabeling during a Prometheus scrape;
- scrape_samples_scraped: total number of samples scraped by Prometheus during scrape operation;
- scrape_series_added: number of time series added by Prometheus during a scrape.


Figure 23: Scrape metrics.

Queries related to the Image Segmentation Metrics row:

- fastapi_inprogress: number of FastAPI requests currently in progress, actively processed by the application;
- fastapi_model_total_requests_total: total number of requests handled by the FastAPI model;
- fastapi_model_segmentation_result_count: count of segmentation results (success/failure) obtained by the FastAPI model.
- fastapi_model_segmentation_result_count_percentage: percentage of successful segmentation tasks compared to the total segmentation tasks, indicating the model's accuracy.

The dashboard was saved as a json file and uploaded to the dashboards folder in the project repository.

## 7.6 Cloud Deployment on Okteto

For the deployment on Okteto, we used the same Docker Compose configuration as in the previous milestone. Okteto is capable of recognizing Docker Compose configurations and converting them into a Kubernetes configuration file, which it uses behind the scenes for the creation and deployment of the necessary services. To accomplish this, it was sufficient to log in via the command line to an enabled account and launch the command okteto up. It is also possible to directly import the repository via the Okteto dashboard, provided the necessary permissions are granted. In our case, two pods were created, one for the frontend application and one for the backend application. These were published through the Okteto SaaS service at the following addresses:

- https://app-albertovalerio.cloud.okteto.net/
- https://api-albertovalerio.cloud.okteto.net/

Furthermore, to optimize storage (limited to 5GB in the free version), a .stignore file was defined. In this file, only the paths strictly necessary for the operation of the services were enabled, not the entire volume of images managed by the model during preprocessing and training.

## 7.7 Monitoring with Better Uptime

Better Uptime is a monitoring platform that provides real-time alerts and status pages, ensuring that teams are always informed about the health and performance of their systems. We incorporated monitoring with Better Uptime into our project by creating an account on Better Uptime's platform and entering our website URL into the Better Uptime dashboard to initiate the monitoring process. This allowed Better Uptime to start tracking the uptime and performance of our website.

The following features of Better Uptime have been utilized in this project:

- Uptime Monitoring: Better Uptime checks the availability of our website every minute from multiple locations around the globe. This helps us ensure that our website is accessible to users worldwide.
- Alerts: We have configured Better Uptime to send alerts via email whenever the website is down. This allows us to respond quickly to any issues.
- Status Page: We have set up a public status page with Better Uptime and added it to the project's README page on github. This page provides our users with real-time information about the operational status of our website.

In particular, the Status Page serves as a comprehensive platform for users to stay informed about the website's status. It features three tabs:

1. Status: Provides real-time updates about the operational status of the website.
2. Maintenance: Informs users about any ongoing or scheduled maintenance activities.
3. Previous Incidents: Offers a historical record of past incidents, providing transparency about the website's performance and uptime.

Overall, the integration with Better Uptime is a significant aspect of this project as it offers essential tools for monitoring the website's uptime and performance, contributing to the overall user experience.

## 7.8 Load Testing with Locust

Locust is an open-source load testing tool used to evaluate the performance of web applications under stress. It simulates users interacting with the application to identify potential bottlenecks and performance issues, ensuring that the system remains responsive and stable under heavy load.

Five different tasks were implemented:

1. Main Endpoint Test: Verifies the server's ability to handle requests to the main endpoint.
2. Get Samples Test: Assesses the functionality of fetching a limited number of images.
3. Upload Image Test: Tests the server's capacity to handle image uploads.

4. Predict Image Test: Focuses on the prediction functionality of the application.

5. Get Metrics Test: Evaluates the server's performance in computing and returning metrics.

Different weights were assigned to each task. Assigning weight arguments to tasks in a Locust load test script is an important step for realistically simulating user behavior, since the weight determines how often a particular task is picked relative to other tasks.

- The main_endpoint task was assigned a weight of 1. The main endpoint (/) is typically the entry point of an application and is likely to be hit frequently. However, it might not be as frequently accessed as specific functionality endpoints once a user is actively engaged with the application. Assigning a lower weight makes sense because, after initial access, users will likely navigate to more specific parts of the application.

- The get_samples task was assigned a weight of 3. This task fetches images from the server and can be considered a common action if users regularly view different images. The higher weight compared to the main endpoint reflects the assumption that once users are in the application, browsing images is a more frequent activity.

- The upload_image task was assigned a weight of 2. Image upload is a critical but potentially less frequent operation than viewing images. Users might spend more time browsing or analyzing images than uploading new ones. Thus, this task should have a moderate weight, indicating it's a regular but not the most common activity.

- The predict_image task was assigned a weight of 2. Making a prediction is a core functionality of the application, but might not be used as often as simply viewing images. It's reasonable to assume that for every few images viewed, a prediction might be made on one of them. A weight of 2 strikes a balance between frequent and occasional use.

- The get_metrics task was assigned a weight of 1. Requesting metrics might be less frequent compared to other tasks like viewing or uploading images. It's often an action taken after several other interactions (like uploading or predicting images). Therefore, a lower weight is justified as it's likely an action taken occasionally, perhaps for detailed analysis or review of a prediction.

The implemented locustfile also has the following characteristics:

- Error Handling: The locustfile was designed with robust error handling, ensuring that any non-200 status codes or unexpected responses from the server triggered explicit error messages. This feature is critical for quickly identifying and addressing issues during the testing phase.

- Random Image Selection in Tasks: To enhance the robustness of our load tests, some tasks in the locustfile were designed to select images randomly. This approach prevented the caching mechanism from skewing the test results, providing a more accurate representation of the server's performance under varied and unpredictable conditions.

We devised four different load tests in order to test the system under different conditions:

1. Baseline Test: small number of users (5) with a low spawn rate (1 user per second) for a moderate duration (10 minutes). This test is useful to ensure that the system is working correctly and can handle a minimal amount of traffic. It can help identify any glaring issues before starting more intensive testing.

2. Moderate Load Test: moderate number of users (20) with a moderate spawn rate (2 users per second) for a longer duration (30 minutes). This test is designed to simulate a hypothesized typical load on the system. It can help understand how the system performs under normal conditions and identify any performance issues that might affect the user experience.

3. High Load Test: high number of users (50) with a high spawn rate (5 users per second) for a longer duration (30 minutes). This test is designed to stress the system and see how it performs under heavy load. It can help identify any performance bottlenecks and understand how the system behaves when it's close to its capacity.

4. Longevity Test: moderate number of users (20) with a moderate spawn rate (2 users per second) for a very long duration (2 hours). This test is designed to check for issues that might only become apparent over time. It can help ensure that the system is stable and can handle a sustained load for a long period.

The different load testing configurations are summarized in Table 3.

| **Test** | **Number of users** | **Spawn rate** | **Duration** |
|---|---|---|---|
| Baseline Test | 5 | 1 | 10 minutes |
| Moderate Load Test | 20 | 2 | 30 minutes |
| High Load Test | 50 | 5 | 30 minutes |
| Longevity Test | 20 | 2 | 2 hours |

Table 3: Load Testing Configurations

During the first run of the baseline test, we encountered a high failure rate. After analyzing the failures encountered by Locust, we decided to partially refactor our server code in order to improve the server's efficiency and its ability to handle multiple users. Two main improvements were made:

- Image Saving Optimization: In the previous version of the server code, all images were read into memory and then saved to the TEMP_PATH. In the new version, each image is checked if it already exists in the TEMP_PATH. If it does not, only then is it read into memory and saved. This reduces unnecessary I/O operations and memory usage, especially when dealing with a large number of images.

- File Upload Improvement: In the previous version, every uploaded file was saved with the same name 'upload.jpg', which could cause issues when multiple users are uploading files simultaneously. In the new version, a counter is added to the filename if a file with the same name already exists in the TEMP_PATH. This ensures that every uploaded file has a unique name, thus preventing any potential file conflicts.

We were able to verify that implementing these optimizations was beneficial when we re-ran the baseline test after these changes and obtained a rounded 0% failure rate, with only a single failure out of 981 requests, as can be seen in Figure 24.

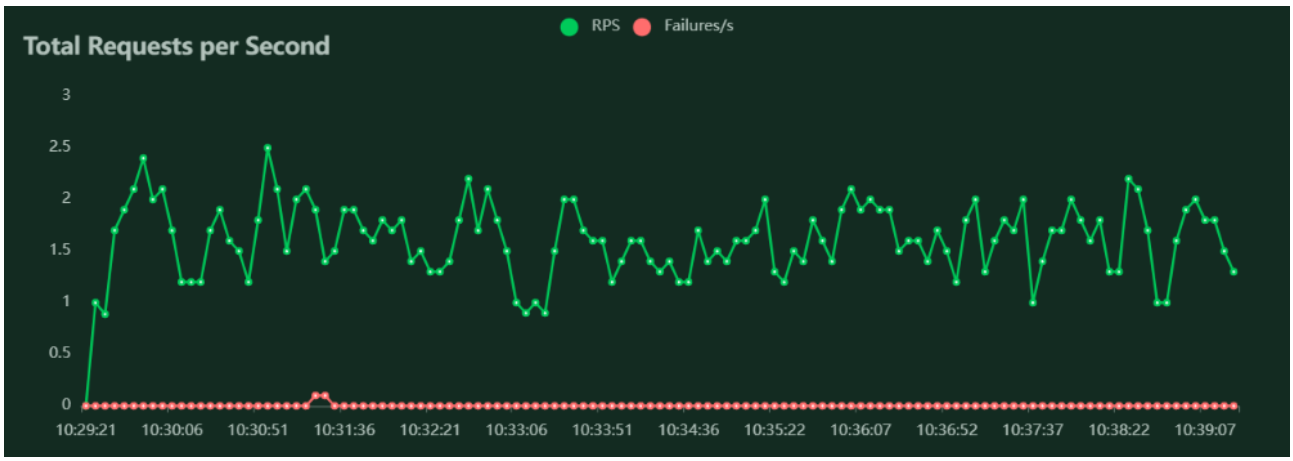


Figure 24: RPS vs. Failures for the Baseline Test.

Following this, we performed the moderate load test, which resulted in a rounded failure rate of 0%, with 11 failures out of a total of 2873 requests, as seen in Figure 25.

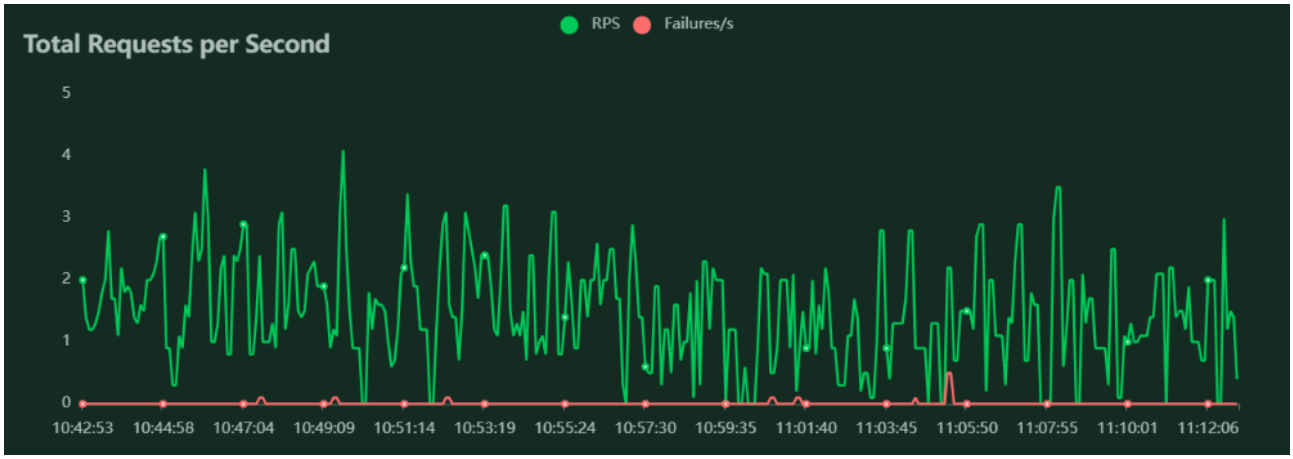


Figure 25: RPS vs. Failures for the Moderate Load Test.

The number of failures is obviously higher than for the baseline test, but this is expected given the quadruple number of users, the double spawn rate and the triple testing time.

The most intensive test that we conducted was the high load test, with 50 users and a spawn rate of 5. During this test, a 14% failure rate was detected, as can be observed in Figure 26. This is much higher than the previous two tests, but is still a good result if we take into consideration the high number of users and the high spawn rate, in addition to the fact that the system is hosted using Okteto's free trial.

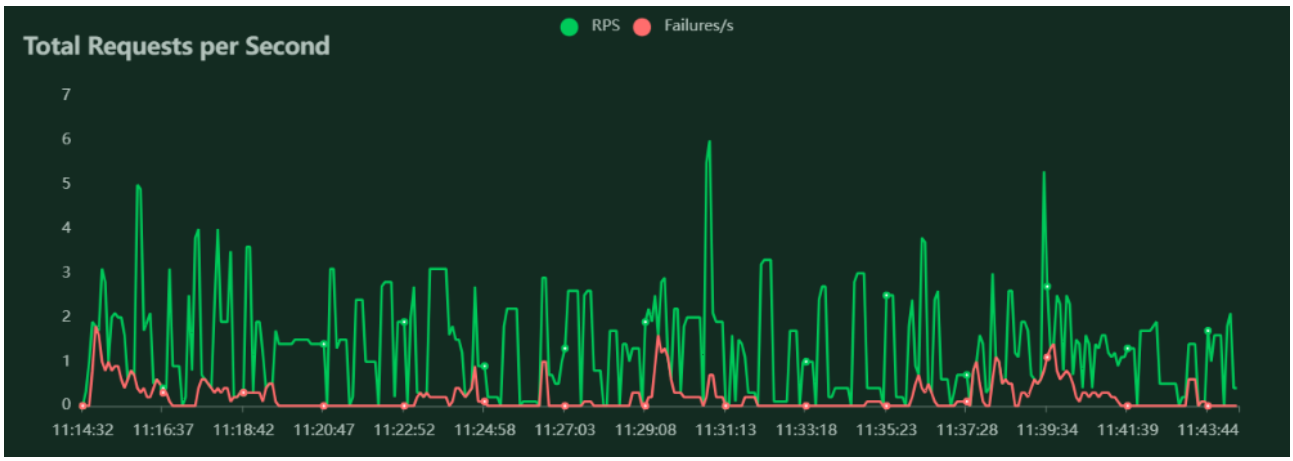


Figure 26: RPS vs. Failures for the High Load Test.

Lastly, the longevity test was conducted. During this test, a 6% failure rate was detected, which is to be expected given the long duration of the test and showcases generally satisfactory performance. The chart for this test is showcased in Figure 27.

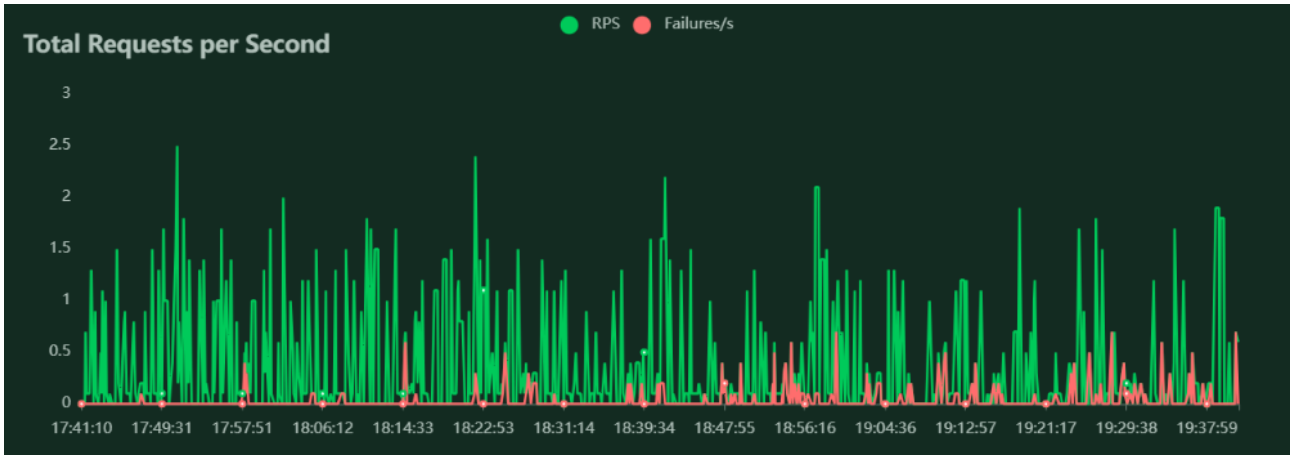


Figure 27: RPS vs. Failures for the High Load Test.

The results of the conducted tests are summarized in Table 4.

| Test | Users | Spawn rate | Duration | Failure rate |
|---|---|---|---|---|
| Baseline Test | 5 | 1 | 10 minutes | 0% |
| Moderate Load Test | 20 | 2 | 30 minutes | 0% |
| High Load Test | 50 | 5 | 30 minutes | 14% |
| Longevity Test | 20 | 2 | 2 hours | 6% |

Table 4: Load Testing Results

As can be seen from the results of the conducted tests, the system performed very well during the baseline test and the moderate load test and moderately well during the longevity test, signifying that it works correctly and that it can perform well under normal conditions without any significant performance issues for an extended period of time. The high load test, with its lower performance, was useful for understanding how the system behaves when it's close to its capacity. Performing load testing of our system helped us get an idea of our

system's capacity for handling concurrent users and determine the performance threshold under varying load conditions. This information is important for optimizing our system for peak performance and ensuring a seamless user experience, even during periods of high traffic. It also aids in identifying potential bottlenecks and areas that require further improvement or scaling. The results of the conducted tests were uploaded to the reports/locust folder of our repository, including the Locust HTML report and the CSV files reporting requests, failures, and exceptions for each test.

### 7.9 Corrections

As recommended during the review of the work completed for this milestone, we implemented an automated workflow using GitHub Actions that periodically checks for data drift using the Alibi Detect library. The script first extracts a vector representation from the images (as described in point 2. Drift Analysis with Alibi), then saves this numerical representation to a timestamped file for comparison over time. In addition, a log file is generated at each execution, indicating whether the check was successful or if data drift was detected. This log file, in addition to being downloadable as a workflow artifact, is pushed into the repository to maintain a temporal trace of all data checks performed. Finally, if data drift is detected, the workflow is forced to terminate with an error (failure), to immediately highlight the presence of a problem that requires further attention and checks.

## 8 Conclusion

The project undertaken in the "Software Engineering for AI-Enabled Systems" course for the academic year 2023-2024, has been a detailed journey in enhancing an existing AI model for crop segmentation. The presented report detailed the systematic progression through various phases, each critical to the development of a robust, efficient, and reliable AI-enabled system. These stages, ranging from Inception to Monitoring, ensured the integration of the existing Computer Vision model with modern software engineering methodologies and tools. This approach not only strengthened the model's efficiency and reliability but also provided valuable insights into the nuances of working with pre-existing AI systems. The project underscores the importance of software engineering principles in the realm of AI, demonstrating that the development of AI-enabled systems extends beyond the initial creation of the model to include ongoing enhancements, quality assurance, and adaptability to new challenges and environments.