CropSegmentation: M4

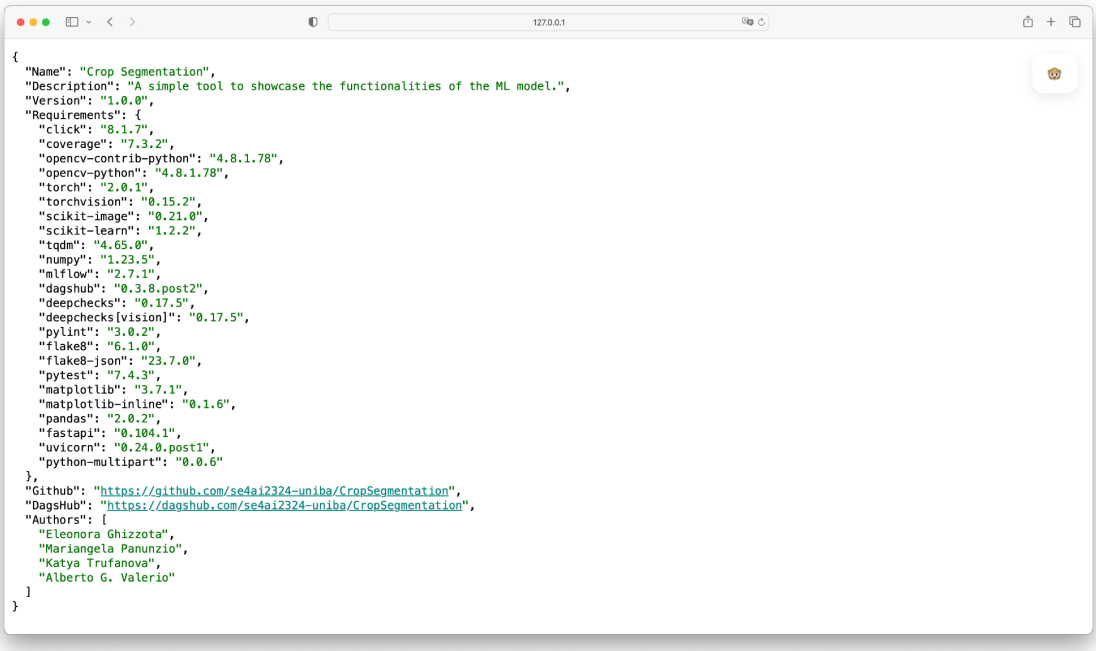
1. M3 Corrections

After the feedback and instructions received during the presentation of the third milestone, the presentation documents for the previous milestones (M2 and M3) were added to the repository, as part of the artifacts of the project. They can be found in the ‘references’ folder, along with the document for the current presentation.

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.



- **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.

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.

4. API Documentation with ReDoc

FastAPI provides two distinct tools for API documentation: Swagger UI and ReDoc. Each of these tools offers unique features that cater to 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. 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.

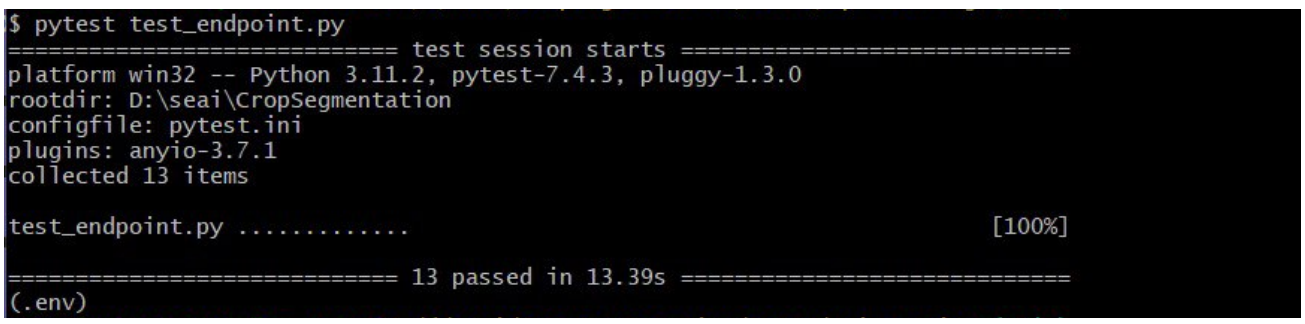
Below is a detailed overview of the implemented tests.

- **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.

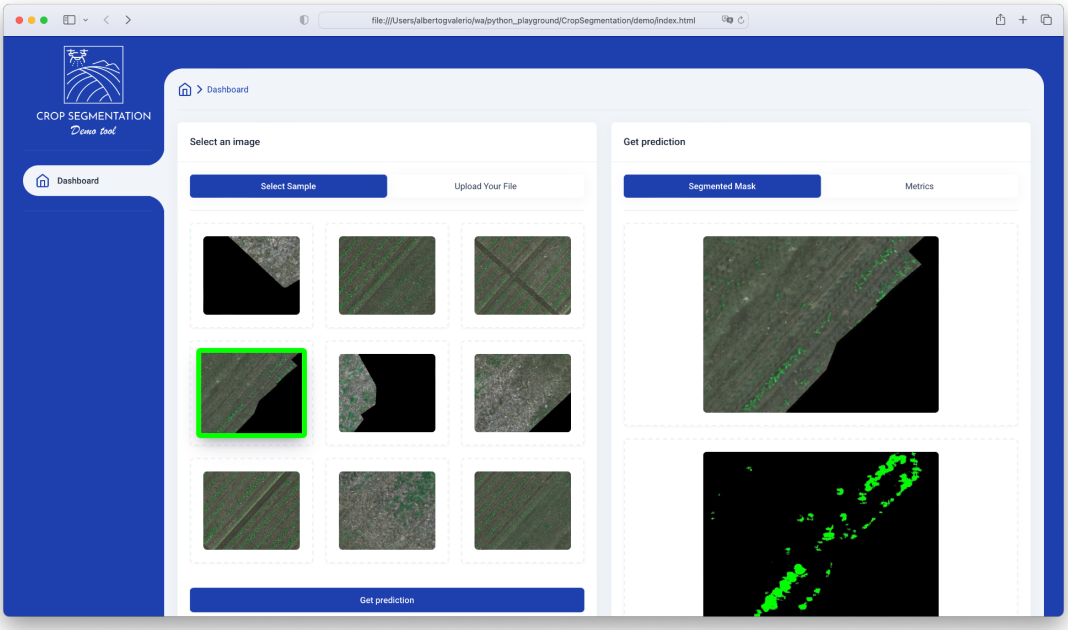
The screenshot below showcases the successful execution of all the implemented tests:



6. Demonstrating 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.



- **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.