APSAP_Sherd_Matching_Tool

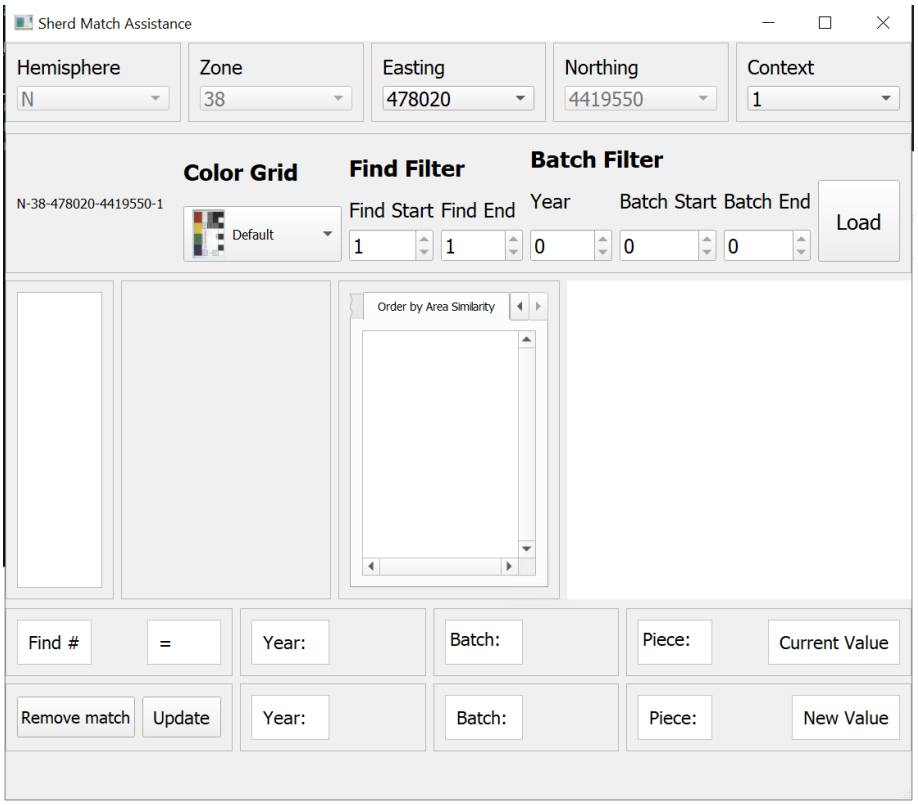
Development Guide

Introduction

This programmer guide is intended for programmers. It is helpful if you want to understand the theory behind the application and have a high level understanding of the general architecture of the application. As the code itself contains precise annotations and comments that serve as explanations, this document explains things in a higher level view.

This document contains the following sections

1. Technical Overview
2. Architecture Overview
3. Application Objectives
4. Important folder paths
5. Speed Optimisation
6. Measurement
7. Similarity Calculation
8. Code Structure
9. Debugging
10. Training neural networks



Technical Overview

This section outlines the programming language, libraries and frameworks used to develop this application.

The programming language used for developing this application is **Python**. Python is an interpretive language that runs on an environment installed with a **Python Interpreter.**

The Python environment used for this application is **Anaconda**. **Anaconda** allows the user to create a safe Python environment to install the specific Python version and the specific Python packages for the application. The Anaconda environment has been set and installed in the folder **E:\tools\PieceReg\envs\piecereg**. Every execution of this program will use this folder as the environment when running the application.

The GUI environment used for this application is **PYQT**. The specific way to create GUI interface using PYQT is via the use of **QT creator. QT Creator** is a no-code GUI design framework specifically used for creating graphical interfaces for QT applications.

Apart from the inherent data structures in Python, the application also uses **Numpy** extensively to inter-operate different Python libraries and frameworks.

As the application has to deal with 3D models in the point-cloud format**,** the application uses the library **Open3D** to process the 3D models. **Open3D** allows the application to show the 3D model in the viewport for matching, and allows for capturing the 3D model's pixels to measure the pixels.

As mentioned in the user guide of APSAP, the application measures pixels of jpegs. It does so by using a trained neural network in **Pytorch,** the model architecture being **MaskRCNN**. It is the crux of this application and allows dramatic improvement of matching speed.

For measured data of the 3D models, a cache library in Python called **Diskcache** is used. It saves Pythonic variables directly to the disk and it allows access from multiple instances of the application at the same time. It makes sure that each 3d model is measured only once and saved in the **cache** folder so that subsequent measuring is instantaneous.

Architecture Overview

This application uses a simplified Model-Presenter-View architecture.

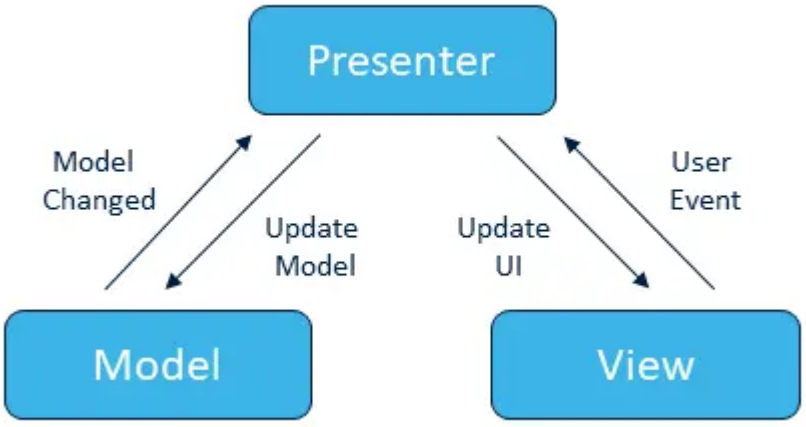
One can gain a better understanding of MVP by understanding the individual concepts of model, view and presenter respectively. Here's a short exposition of these concepts.

First, **Model** is related to the data aspect of the application. All applications use data in one way or another. APSAP uses data in various ways. First, it has to use spatial or graphical data in the forms of **JPG or PLY,** the file formats of photos and 3D models. Second, it has to deal with **JSON** files related to basic configuration and paths. Third, it has to deal with data in the **database**. Fourth, it has to save data into the folder's cache folder. All these functionalities related to data are in the **Model** portion of the application.

Next, **View** is related to the graphical interface of the application. There are a few elements in this section. First, it has Python code that loads the GUI elements. Second, it has the **.ui** file(s) that the Python code loads to configure the user interface. Note that **.ui** files are created using **Qt creator.** Third, it contains small static icon files.

Third, **Presenter** is related to most of the code not fitting in the **Model and View** sections. It mostly contains code of **callback functions** that get activated when the user interacts with the graphical interface.

The actual implementation of the architecture will be better spelled out in the section about the project folder structure.



Application Objectives

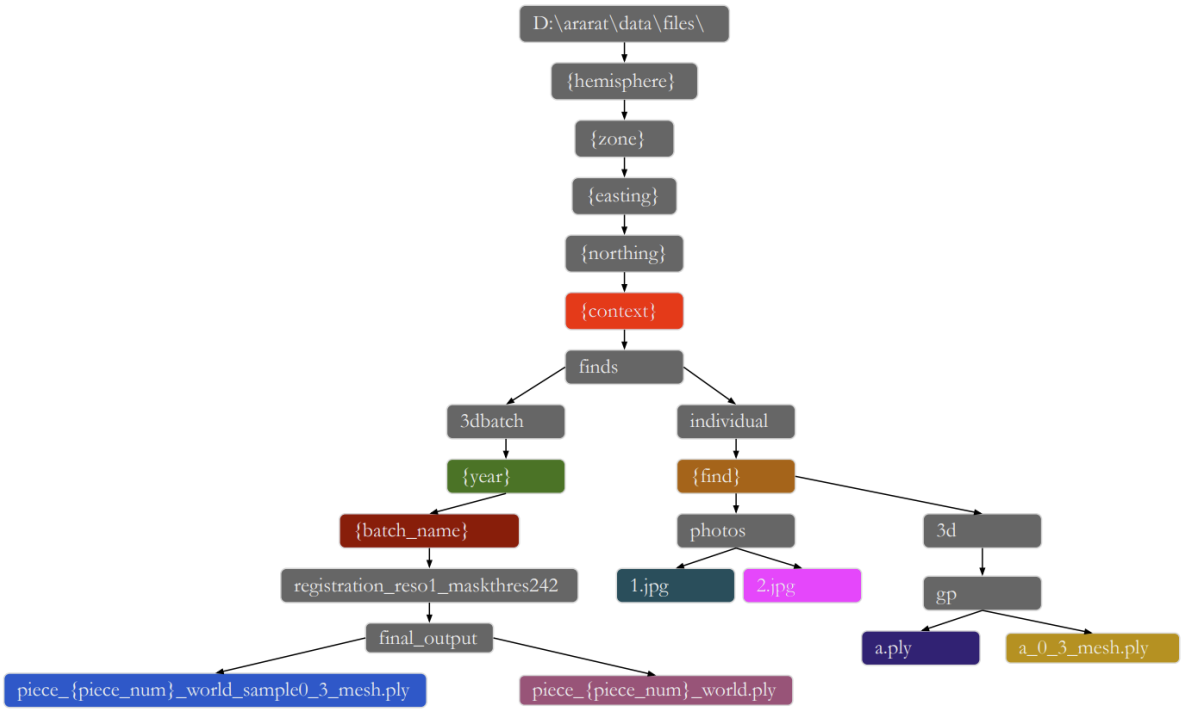
Behind the complexity of the code and architecture, the application has two very specific aims:

1. Updating the database such that a certain find at a certain context is matched with the correct 3D models with the correct year, batch and find numbers.
2. Copy the 3D models from the matched 3D model folders to the matched find folder
    a. An additional task is to fix the 3D model headers so that it can be opened by the application Gilgamesh.

The application does all these when you click on the update button.

Important folder paths

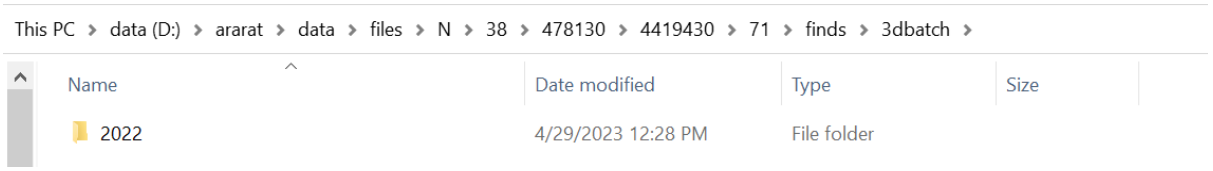
As finds and 3D models are in folders, it is important to know all the important folder and file paths this application will interact with. The following diagram demonstrates the folder paths for all important paths, whose background color is set to be not gray.



Context: A specific context which we want to investigate.
Example:
D:\ararat\data\files\N\38\478130\4419430\71

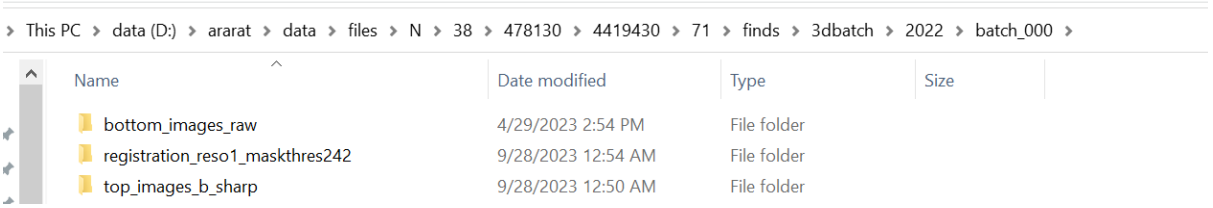
Year: A folder whose name is a specific year(e.g. 2022), which contains batch folders inside that year

Example:



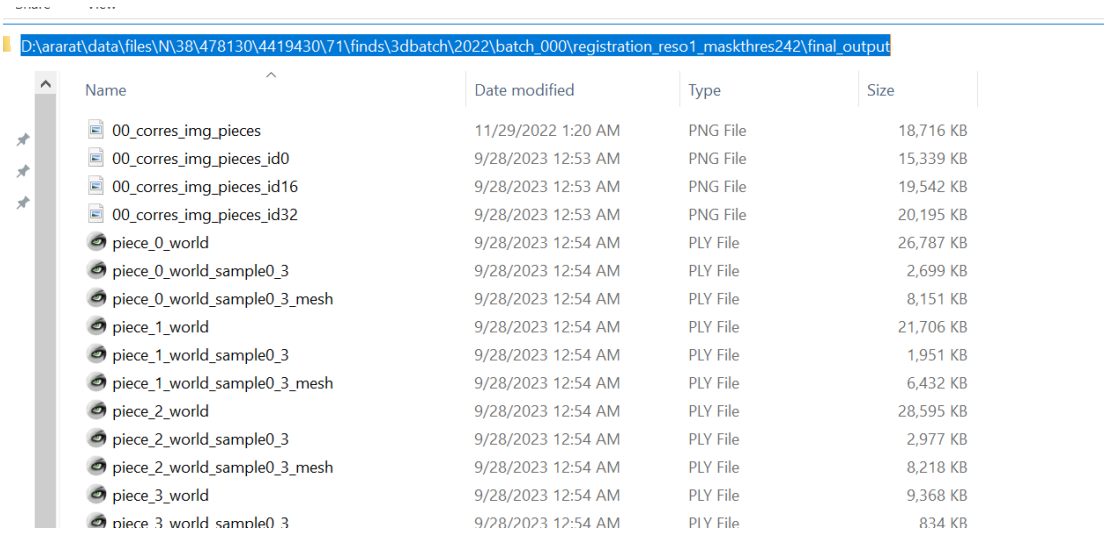
Batch: A specific batch folder that contains 3D model content inside

Example



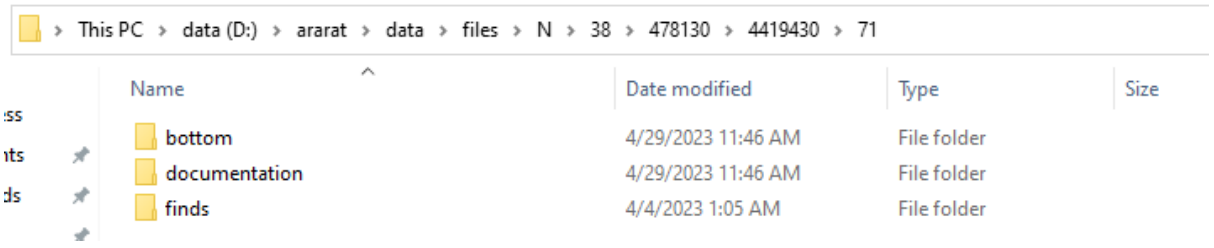
Downsampled 3D model and Original 3D model: The original sized 3D model and the smaller one inside a batch folder. A batch folder usually have many such models

Example



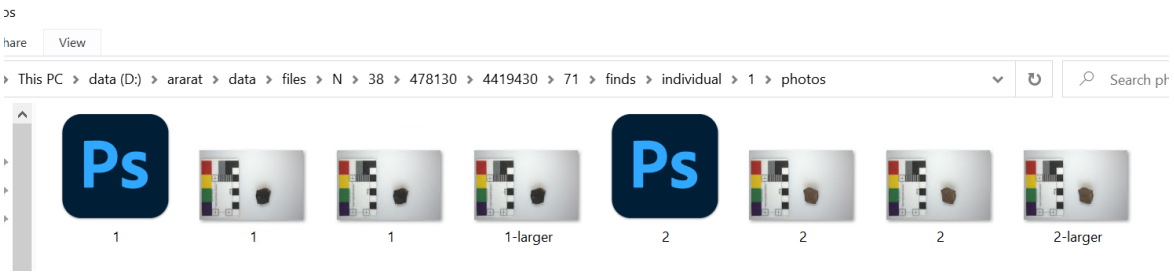
Find: The folder of a specific find in the context

Example



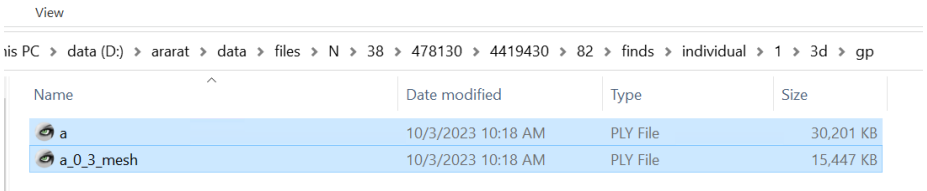
1.jpg and 2.jpg: Two images that identify what the find looks like.

Example



a.ply and a_0_3_mesh.ply: These files are the 3D models of the find. They get copied here when a matching happens.

Example



Now we know the important paths, we know more precisely what the application does when it does matching:

1. It updates in the database the row of the Find with the updated year-batch-piece information in Downsampled 3D model and Original 3D model.
2. It copies the 3D models from the path Original 3D model and downsampled 3D model to the paths a.ply and a_0_3_mesh.ply

Speed Optimization

This section talks about how the application optimizes the matching process. It first talks about the matching time required for matching without using any technique. Then it talks about the matching time improvement using the techniques.

Average time of matching naively

Suppose the user has to match 100 models with 100 finds. Suppose it takes 1 second to go through 1 model. Given a find, the best scenario is that the first model is the find we want, the worst scenario is that the last model on the list is the find we want. In the former case, we have to go through 0 models, in the later case we have to go through 100 models. So on average, we have to go through (100-0)/2 =50 models for each find. So we have to go through a total of 50 * 100 = 5000 seconds to do the matching given 100 finds.

Average time of matching by ignoring matched finds

Now suppose we match in the same way but this time we ignore finds that have been matched before. In particular, models that have been matched are marked as red. Given this scenario, our matching will look something like this:
For the first find, we have to go through on average,100/2 = 50 models,
For the second find, we have to go through on average,99/ 2 = 49.5 models.
For the third find, we have to go through on average, 98 / 2 = 49 models.
…
The number of seconds we need is:
50 + 49.5 + 49 + … + 0.5 + 0 = 2525 seconds. An improvement of a factor of 2.

Matching with an accurate sorted list of similarity

Now suppose that we are able to measure the pixels of the 3D models and 2d images. It takes 1 second to measure 1 jpeg and 1 second to 1 3D model(In practice, it takes less time than that). By measuring the pixels of 200 pictures(front and back images for each find) , it takes a total of 200 + 100 = 300 seconds.

Now, given the measurement, we can calculate the similarities between the 100 3D models with the find we are considering. Now given a highly accurate similarity calculator(which we have), because the 3D model is on the top of the list, we just need to use 1 second to check the 3D model and match it with the find.

This means the total time required for the 100 models is 100 seconds(because it is always on the top of the list in our scenario). With the initial 300 seconds overhead, we need a total of 400 seconds. This is a dramatic improvement compared to the last method!

Measurement

Given that we need to calculate the similarities between 3D models and jpegs, it is necessary for us to measure the pixels in 3D models and jpegs with certain aspects.

Here is a list of the aspects of the pixels that the application measures:

1. Area
2. Width and length of the bounding box
3. Contour

Similarity calculations based on these aspects yield excellent results.

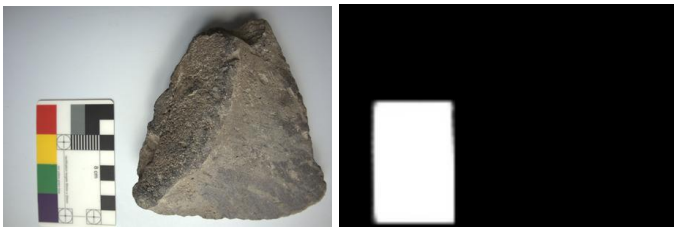
Measurements of jpegs

We need these crucial libraries to measure the pixels

1. Mask R-cnn: A pytorch neural architecture
2. Opencv: a commonly used library for Computer Vision in Python

The first thing we need to do is to isolate the pixels of the ceramics sherd. This can be easily done with a trained neural network model in Masked RCNN.

Secondly, we need to be able to isolate the pixels of the color-grid in the photo



Given these, we can now measure the jpegs.

Measuring the area of jpegs

First, to calculate the actual area of the sherd in the jpeg we count the **number of pixels** of the sherd. We then calculate the **ratio between one pixel in the jpeg and one mm in real life.** We can calculate the ratio because we know the width of the color grid(in this case 53.98mm) and the number of pixels representing the width in the grid(via the neural network model).

Then we can get the actual area of ceramic sherds with the following formula:

Actual sherd Area = num_of_sherd_pixels * (mm-to-pixel_length)^2

Measuring the width-length of jpegs

Second, to calculate the width of the bounding box. We get the vertical and horizontal distance of the sherd in terms of pixels by subtracting bottom pixel location from the top pixel location and the leftmost pixel location from the rightmost pixel location respectively, using the result of the neural network which captures the mask of the ceramic sherd. The width is the shorter distance, the length is the longer distance. Then, given the ratio between one pixel in the jpeg and one mm in real life, we can get the actual width and length by the following formulas:

Actual width = width in pixels * mm-to-pixel_length
Actual length = length in pixels * mm-to-pixel length

Measuring the contour of jpegs

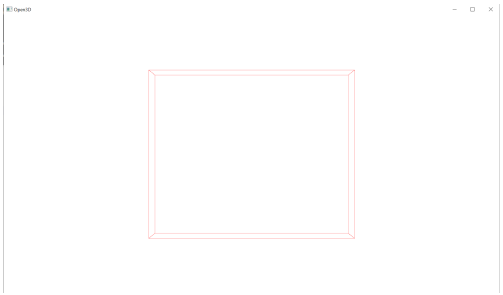
Finally, to get the contour of the ceramic sherd. What we can do is run Opencv's findContours method with the **black and white picture of the masked sherd**. The function returns a list of contours it finds. We return the first one because there is only one contour.

Measurements of the 3D models

To measure the 3D models, we need to use Open3D, a library in Python used for loading, displaying and manipulating 3D models.

The first thing we need to do is to isolate the pixels from the 3D model. What we can do is open the 3D model in Open3D, then capture what it looks like. The background pixels are all pure white, whereas the sherds have variable color pixels.

The second thing we need to do is to isolate the pixels from the bounding box of the 3D model.



Measuring the area of point clouds

To calculate the area, we first count the number of pixels in the captured image, by counting those pixels that are not pure white. Then we calculate the mm to pixels ratio by dividing the actual distance in the 3D space(in mm) by the distance of the red pixels in the bounding box.

So, we have the following formulas:

mm_pixel_ratio
= horizontal_distance_bounding box_in_mm / horizontal_pixel_distance

Area = (num_of_sherd_pixels * mm_pixel_ratio)^2

Measuring the width and length of point clouds

To calculate the width and length of the bounding box, we use the red bounding box's pixels as a reference, and use the formula.

Actual width = width in pixels * mm_pixel_ratio
Actual length = length in pixels * mm_pixel_ratio

Measuring the contour of point clouds

To calculate the contour, we have to mask the sherd area. We can do this by turning the background of the captured from white to black, and the sherd area from colorful to completely white, to simulate the effect of masking using a neural network. Then we can use Opencv's findContours().

Similarity Calculation

Similarity Calculation for the individual components

Given that we have measured the area, width, length and contours for both the jpegs and 3D models. Now we calculate the similarity with respect to area, to width_length and to contour.

For **area and width_length,** we use the following similarity formula

Similarity(a, b) = abs(a - b) / (a + b + 0.0000000001).

This will result in a value between 0 and 1 that indicates how similar two values are. The more similar they are, the closer the value is to 0. The more different they are, the closer the value is to 1. Notice the 0.0000000001 here is to ensure that the denominator is not 0.

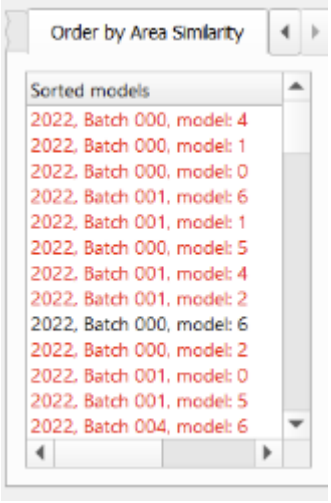
For **contour**, we use Opencv's matchShapes() method. Similarly, a value of 0 indicates an exact match, a value 1 indicates a total lack of similarity.

Total similarity calculation

The total similarity is simply a weighted sum of the individual similarities.

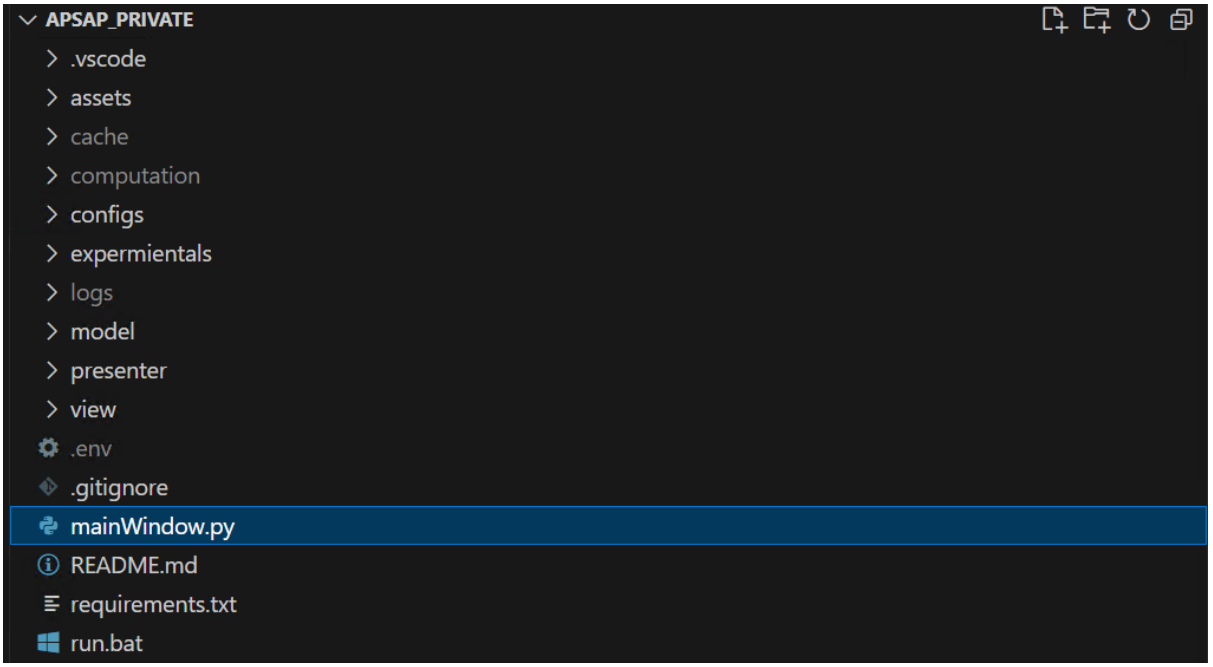
Generating a sorted list of similar 3D models to a set of images

Using the aforementioned methods of measurement and similarity calculation, you can get how similar a 3D model is to a certain find(with its 2 jpegs). We can then sort the 3D models by the similarity. Ultimately, we can get a list of sorted 3D models, where the most similar 3D model is on top of the list. And we achieve our goal of matching time optimisation.



Code Structure

It could be overwhelming for new programmers when they first see the application with its folder structure. This section aims at clarifying it.



There are three types of things here:

1. Folders that help with the application.
2. Folders that contain most of the functionalities.
3. Important files in the root folders.

Folders that help with the application

.vscode:

contains the VSCode editor's configuration files that help with development

assets:

contains static assets the application uses such as the application's icon.

cache:

contains data being cached by the application to save time for long loading part of the application. For example, the measured values of the 3d models are saved here This folder is outside Git's versioning control

computation

contains extra files that help with some computational tasks and neural networks. This folder is outside Git's versioning control

experimentals:

contains files for experimental features

logs:

contains logging files for every instance of usage of the application

Folders that contain most of the functionalities

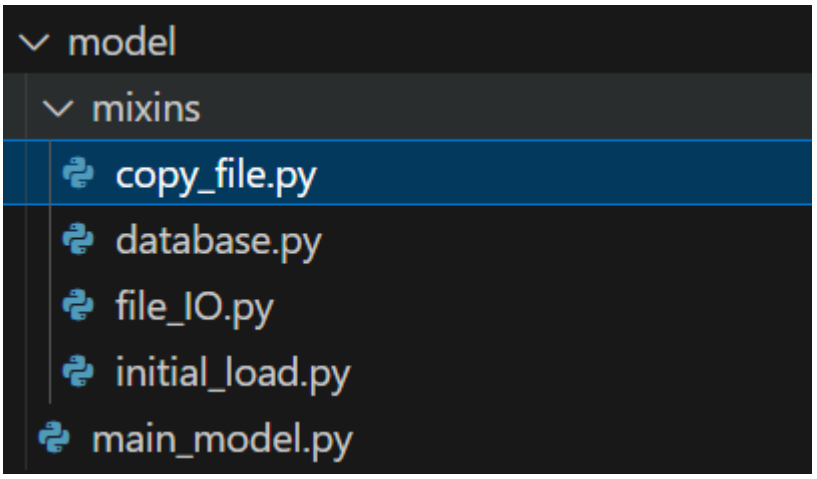
model:
contains the code about the data used in the application
view:
contains the code about the graphical interface of the application
presenter:
contains the code unrelated to model and view, which is mainly the functions that connect between the data(model) and the interactive elements of the GUI(view)

Important files in the root folders

.env:
contains the environment file that includes important environmental information such as the database's info.
.gitignore:
contains files and folders that Git ignores in the folder
mainWindow.py
is the entry point of the application where everything is initialized.
README.md
contains a basic document that explains the application
requirements.txt
contains the Python libraries that this application needs.
run.bat
is a script that initiates the Conda environment and then runs the application.

As most of the functionalities are in the folder **model, view and presenter,** their content will be more thoroughly discussed in this document. Other files and folders are self-explanatory..

Model



The model folder consists of the **main_model.py** and a mixin folder, inside of which are **copy_file.py, database.py, file_IO.py,** and **initial_load.py**

main_model.py:
inherits the individual mixin files, to get their data and methods. The pattern of allowing a main method to inherit the mixins is used also in the **View** and **Presenter** portion of the folder.

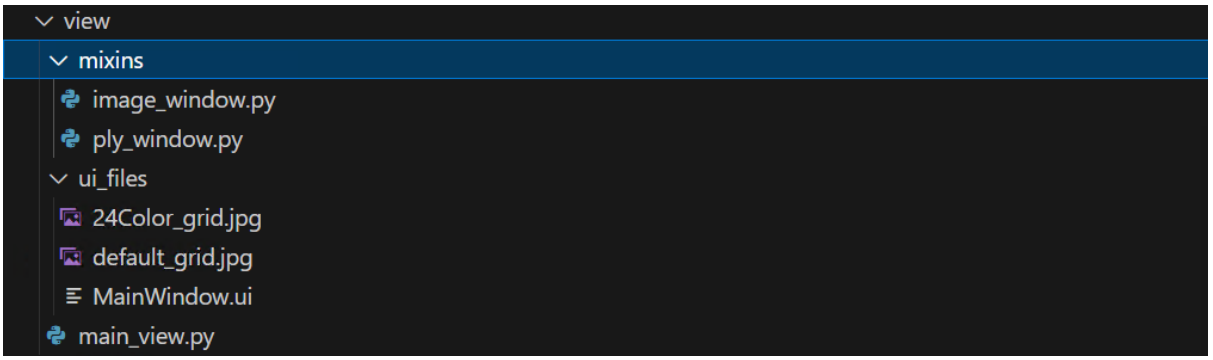
mixins/copy_file.py:
contains the functions related to moving the point clouds from one folder to another

mixins/database.py
contains the functions related to Database access.

mixins/file_IO.py
contains the functions related to the FileIO.

mixins/initial_load.py
contains the functions related to the initialization of the application when it starts.

View



This folder contains the file **main_view.py** and two folders, **mixins** and **ui_files.** Inside the **mixins** folder, we have two files, **images_window.py** and **ply_window.py.** Inside the **ui_files,** we have two jpegs and one ui file.

main_view.py:

inherits all the functions and data in the **mixins** folder, loads data from the **ui_files** folder and instantiates the graphical interface of the application

mixins/image_window.py

contains the code related to clicking the image to create a pop up window with a larger picture

mixins/ply_window.py

contains the code related to the window that displays the 3D model.

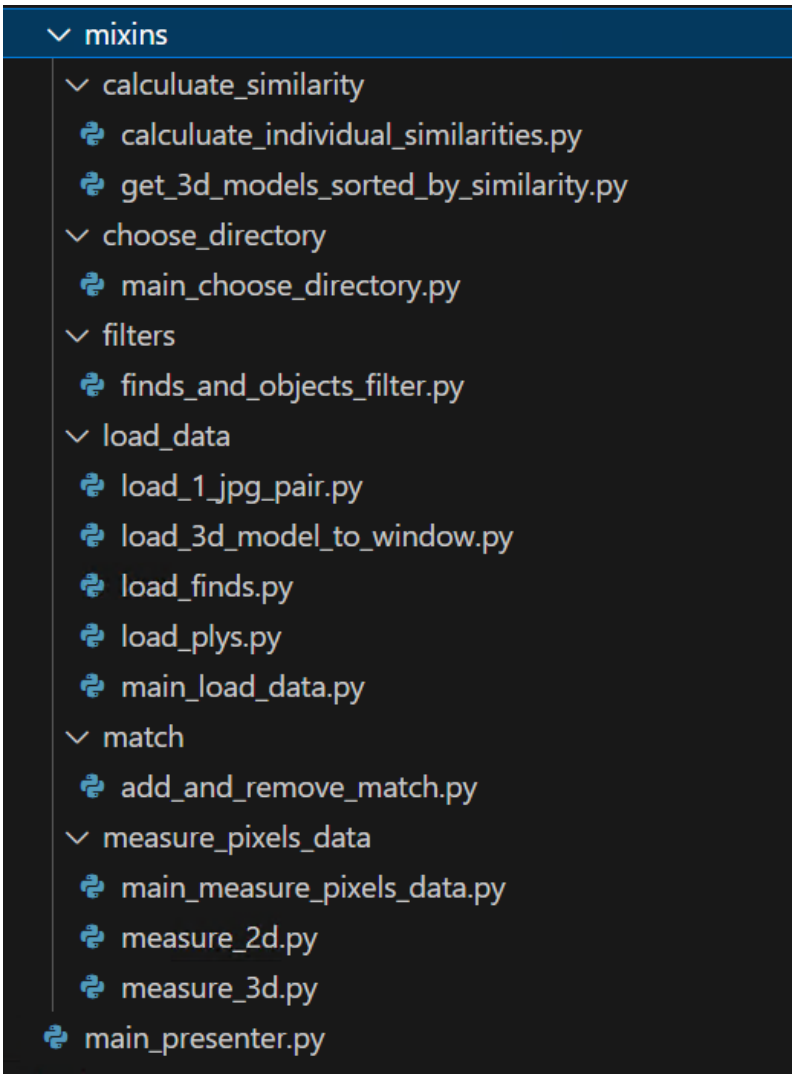
ui_files/24Color_grid.jpg and **ui_files/default_grid.jpg**

are jpeg files that act as icons for the Color Grid List in the GUI

ui_files/MainWindow.ui

is a ui file in the format of the QT Creator. It contains the template of the graphical interface of the application. You can edit this file by opening it in the QT Creator application

Presenter



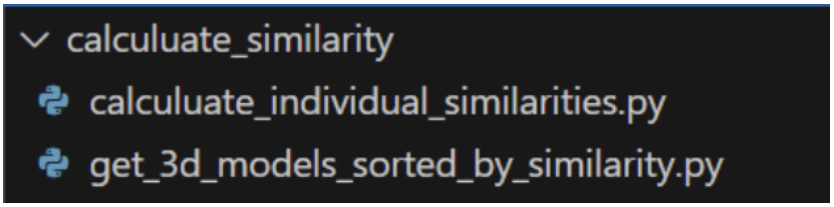
This folder has a lot of functions because it contains most of the code of the callbacks in reaction to interaction with the GUI.

main_presenter:

inherits all the mixins in the mixins folder

mixins/calculuate_similarity

contains the files related to calculating the similarity between a 3D model and a jpeg



mixins/calculuate_similarity/calculuate_individual_similarities.py

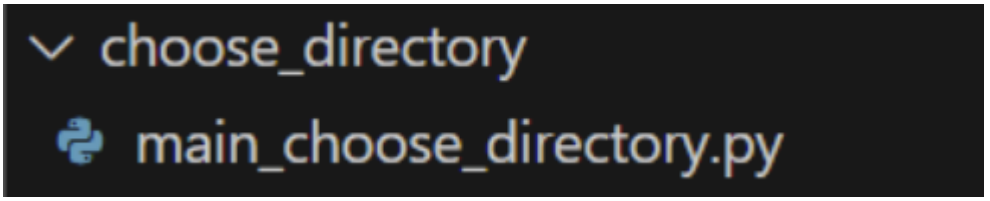
contains individual functions that calculate to similarities in different aspects, such as area, contours, etc.

mixins/calculuate_similarity/get_3D_models_sorted_by_similarity.py

contains the code that supports getting all 3D models sorted by their similarities with respect to a find. It is from here that we are able to fill the list with sorted 3D models

mixins/choose_directory

contains code related to changing the current working directory of the application. The directory is a certain context folder.

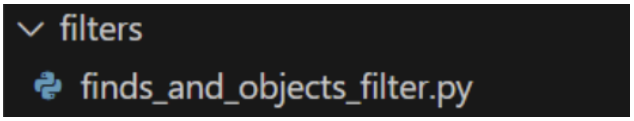


mixins/choose_directory/main_choose-directory.py

contains the code that makes the application change the working directory when the various selection lists of Hemisphere, northing, context, etc are adjusted, and also some code related to getting the current path's hemisphere, northing, context and other info.

mixins/filters

contains the files related to filtering

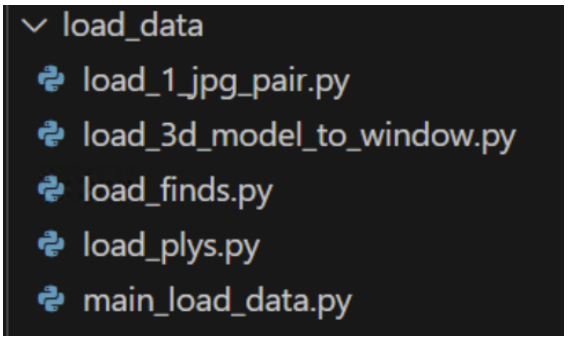


mixins/filters/finds_and_objects_filter.py

contains the files related to filtering 3D models and 2d finds during the matching process

mixins/load_data

contains different files related to loading images, 3D models and other stuff.



mixins/load_data/load_1_jpg_pair.py

contains the callback functions that get activated when you try to load a certain find, and it helps load the 2 jpegs getting displayed, then load the 3D models sorted by similarity with respect to these 2 jpegs.

mixins/load_data/load_3d_model_to_window.ply

contains the the callback function that gets activated when you try load the selected 3D model to the window. It also contains a function that clears the 3D window.

mixins/load_data/load_finds.ply

handles loading all the finds found in a certain context. The finds are populated to the list on the left of the GUI to allow the user to choose a specific find to match.

mixins/load_data/load_plys.ply

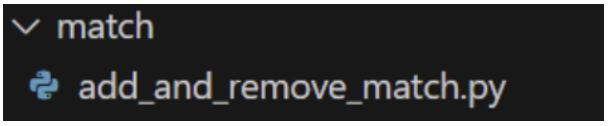
handles loading all the 3D models according to their year, batch and piece number to the selection list. Notice that it is not the list sorted by similarity.

mixins/load_data/main_load_data.ply

loads the various mixins in **mixins/load_data**

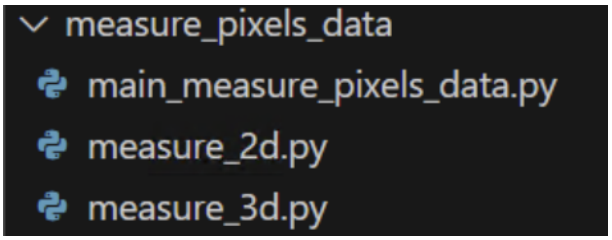
mixins/match

contains files related to matching a find with a 3D model.



mixins/match/add_and_remove_match.py

adds or removes a certain match, updates the database, and moves the 3D models to the matched find folder



mixins/measure_pixels_data

contains files related to measuring the data so that we can calculate the similarity based on the measured data.

mixins/measure_pixels_data/main_measure_pixels_data.py

initializes the neural networks required for measuring pixels, and measures pixels of 3D models and jpegs.

mixins/measure_pixels_data/measure_2d.py

contains the functions that measure all individual aspects of a jpeg(E.g. area).

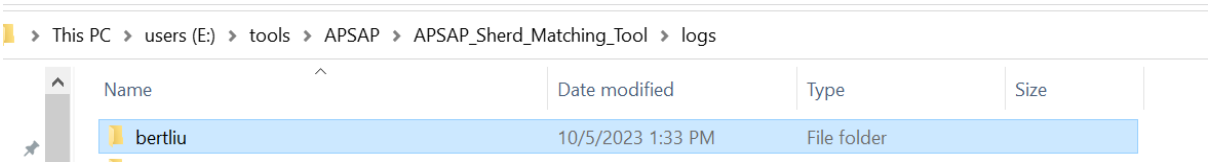
mixins/measure_pixels_data/measure_3d.py

contains the functions that measure all individual aspects of a ply(E.g. area).

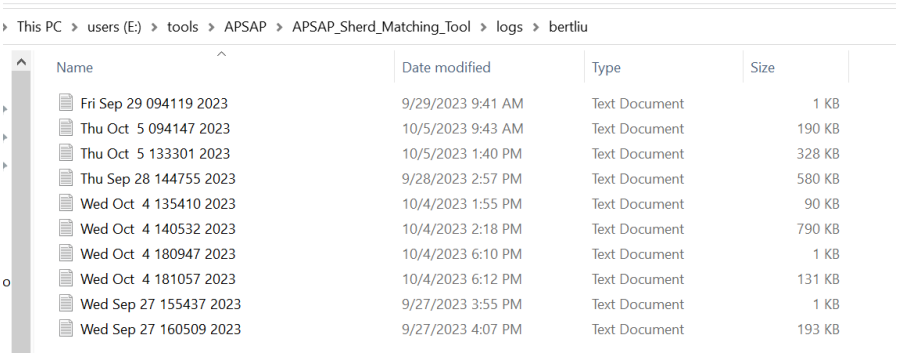
Debugging

In case the application does not work as intended, what you should do is to look at debugging information. The debugging information is located inside the application folder **logs.**

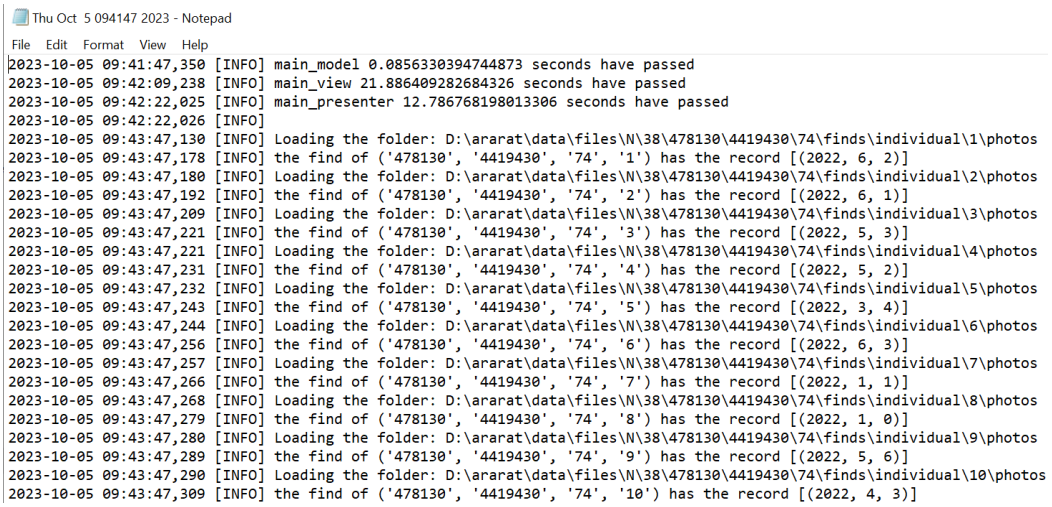
For example, you can see in this folder that there are folders with names of the **user name** of the application.



Inside the folder, you can see log files. Their names indicate the specific date and time the application is opened by the user.



To check the log, open one of the log files, then you can see the precise time and precise logging message for the important operations of the application.



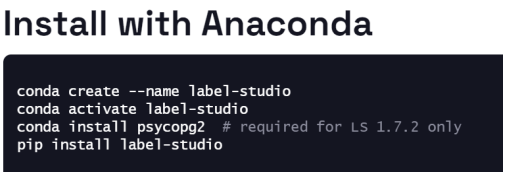
To print extra information, import logging, then use logging.info(message)

Training neural networks

Introduction

As the application uses neural networks to isolate pixels of the ceramic sherds and the color grid, it would help if you know how to train them. This section demonstrates how to train the neural network using Pytorch. You will need to use Label studio to annotate the images. Then you will use a Jupyter notebook to generate the neural network.

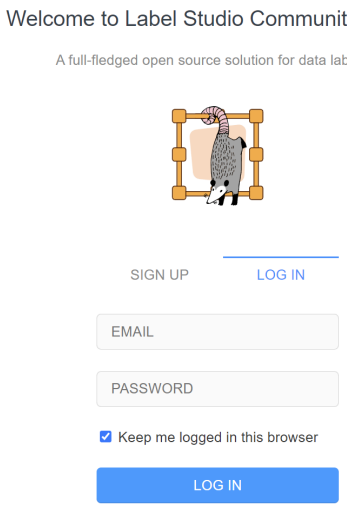
1. Install Label studio
    1. Go to the website https://labelstud.io/guide/install.html
    2. Install Label-studio



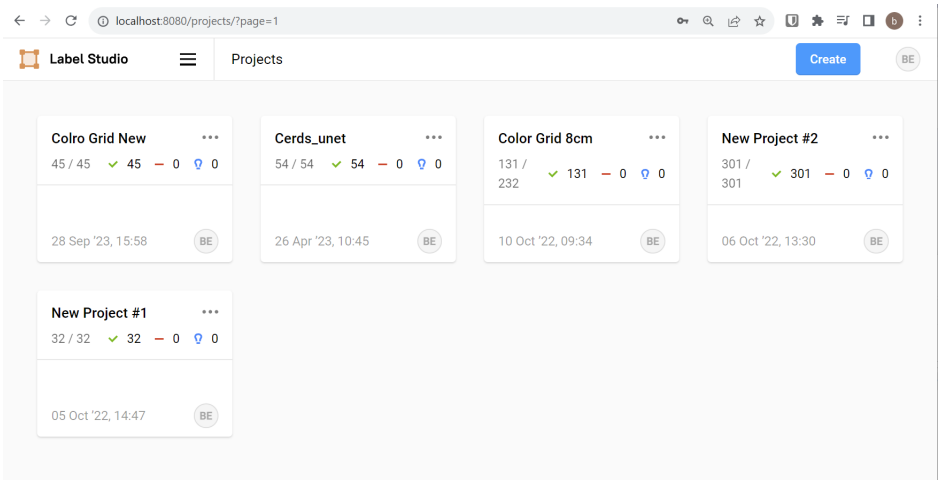
2. Start label studio
    1. run # label-studio start

```
(label-studio) C:\Users\Bert>label-studio start
Current platform is win32, apply sqlite fix
Can't load sqlite3.dll from current directory
=> Database and media directory: C:\Users\Bert\AppData\Local\label-studio\label-studio
Read environment variables from: C:\Users\Bert\AppData\Local\label-studio\label-studio\.env
get 'SECRET_KEY' casted as '<class 'str'>' with default ''
=> Static URL is set to: /static/
=> Database and media directory: C:\Users\Bert\AppData\Local\label-studio\label-studio
Read environment variables from: C:\Users\Bert\AppData\Local\label-studio\label-studio\.env
get 'SECRET_KEY' casted as '<class 'str'>' with default ''
=> Static URL is set to: /static/
[Tracing] Create new propagation context: {'trace_id': 'd5e34a961a2d4e0aac55b2d324adb2ba', 'span_id': '9c4a183b08e74fe9'
, 'parent_span_id': None, 'dynamic_sampling_context': None}
Starting new HTTPS connection (1): pypi.org:443
https://pypi.org:443 "GET /pypi/label-studio/json HTTP/1.1" 200 59014
```

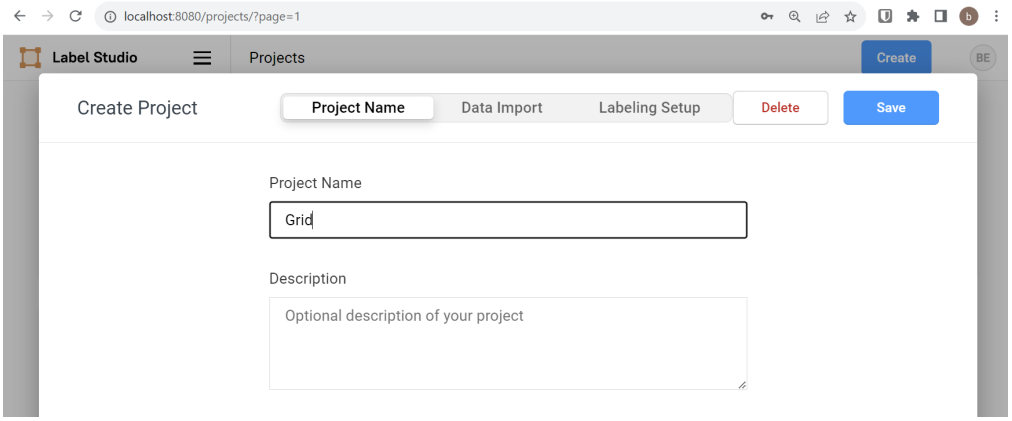
3. Sign up an account and login
    1. Go to http://localhost:8080/user/login/ and login
    2.



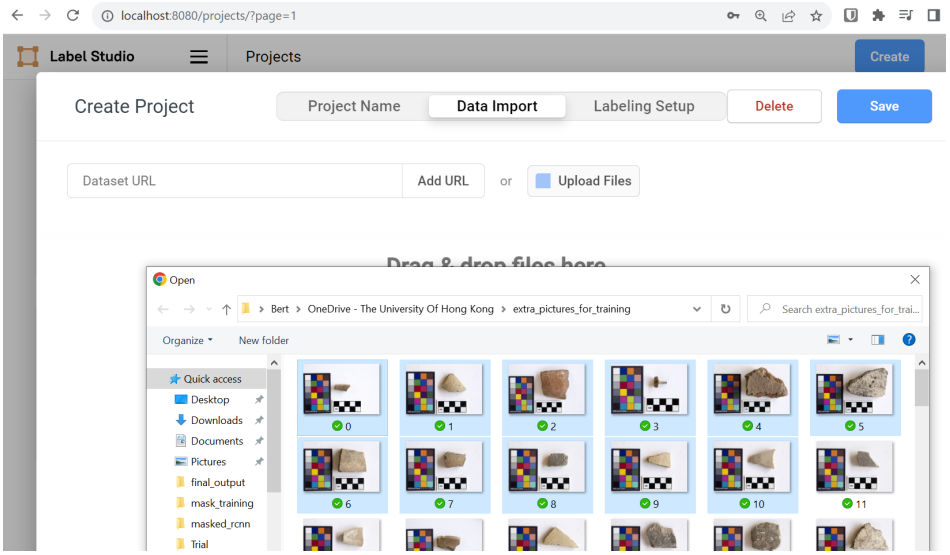
4. Click Create on the first page to create new masks



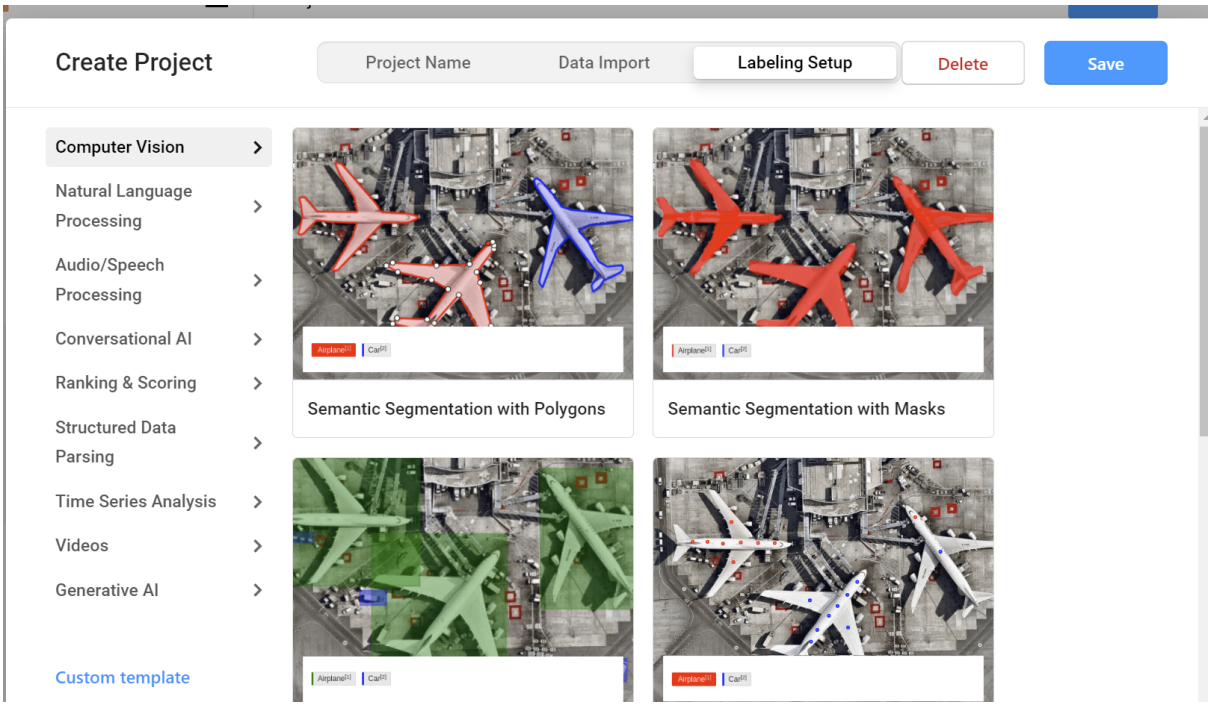
5. Choose the name of the project



6. Choose the images that we will use for training



7. Use a custom template (The blue text on the lower left corner)



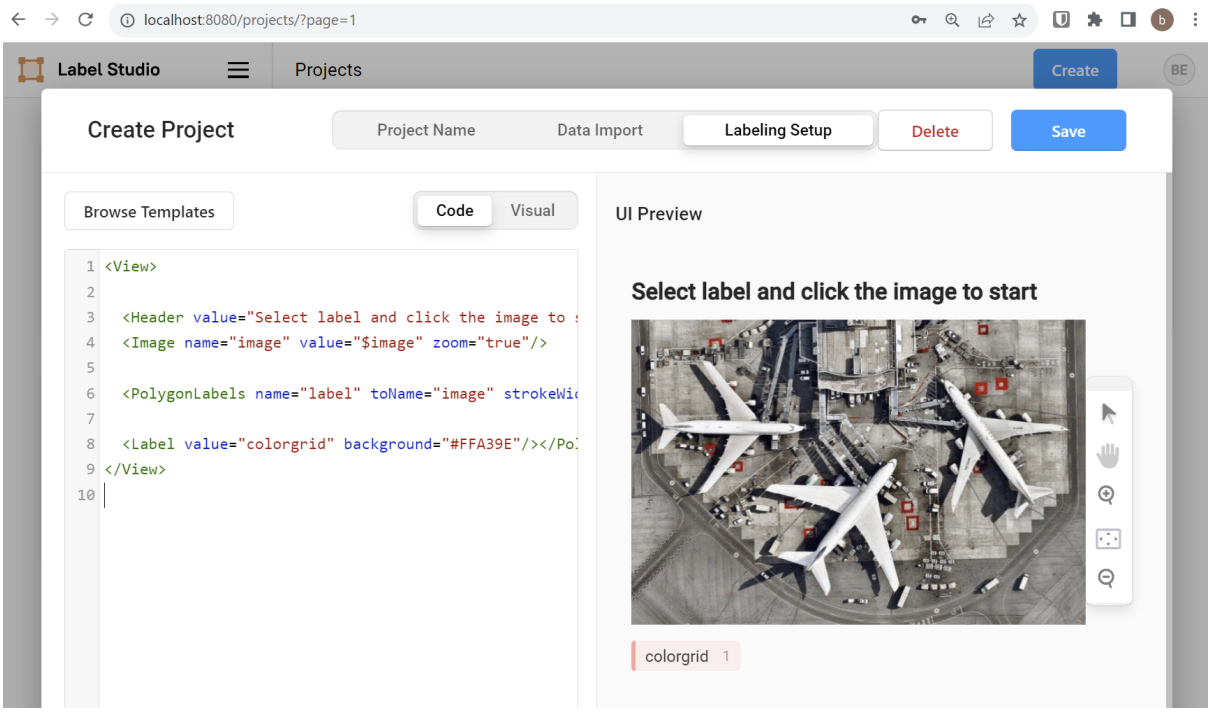
If you want to change a network for the **color grid**, use this code

```
<View>

  <Header value="Select label and click the image to start"/>
  <Image name="image" value="$image" zoom="true"/>

  <PolygonLabels name="label" toName="image" strokeWidth="3" pointSize="small"
opacity="0.9">

  <Label value="colorgrid" background="#FFA39E"/></PolygonLabels>
</View>
```

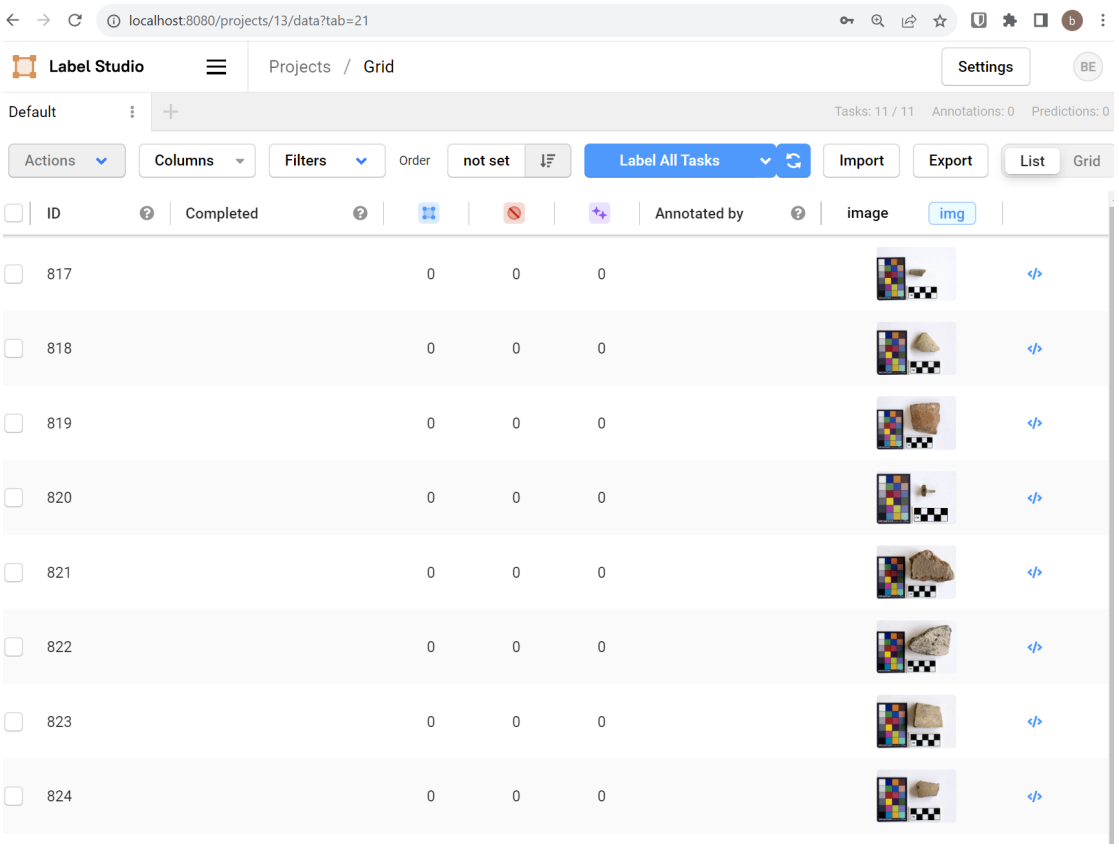
If you want to change a network for the **ceramics**, use this code

```
<View>

  <Header value="Select label and click the image to start"/>
  <Image name="image" value="$image" zoom="true"/>

  <PolygonLabels name="label" toName="image" strokeWidth="3" pointSize="small"
opacity="0.9">


  <Label value="Ceramics" background="#FFA39E"/></PolygonLabels>

</View>
```

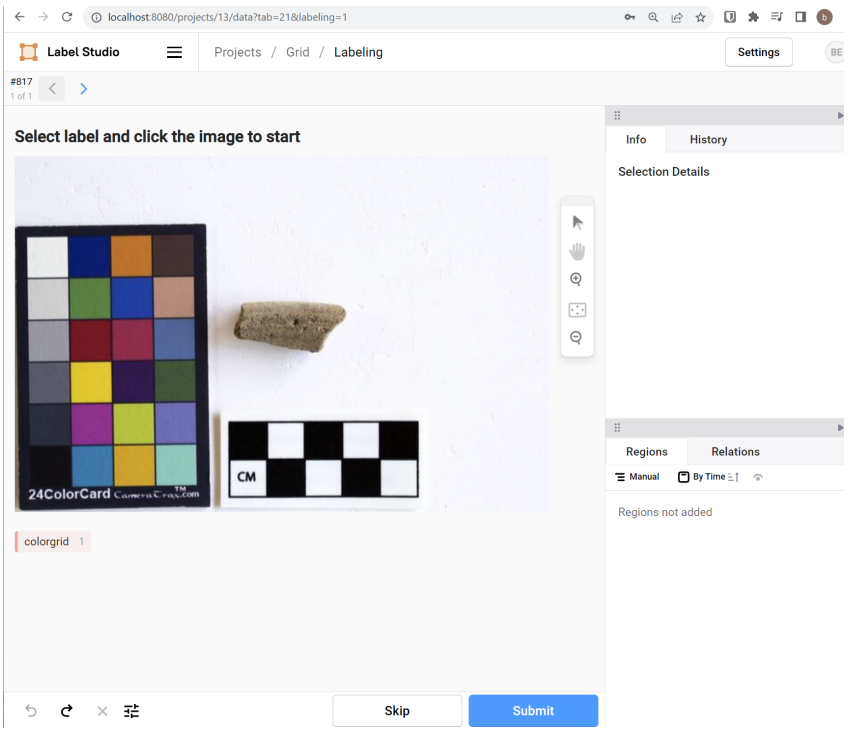
This example uses **color_grid**



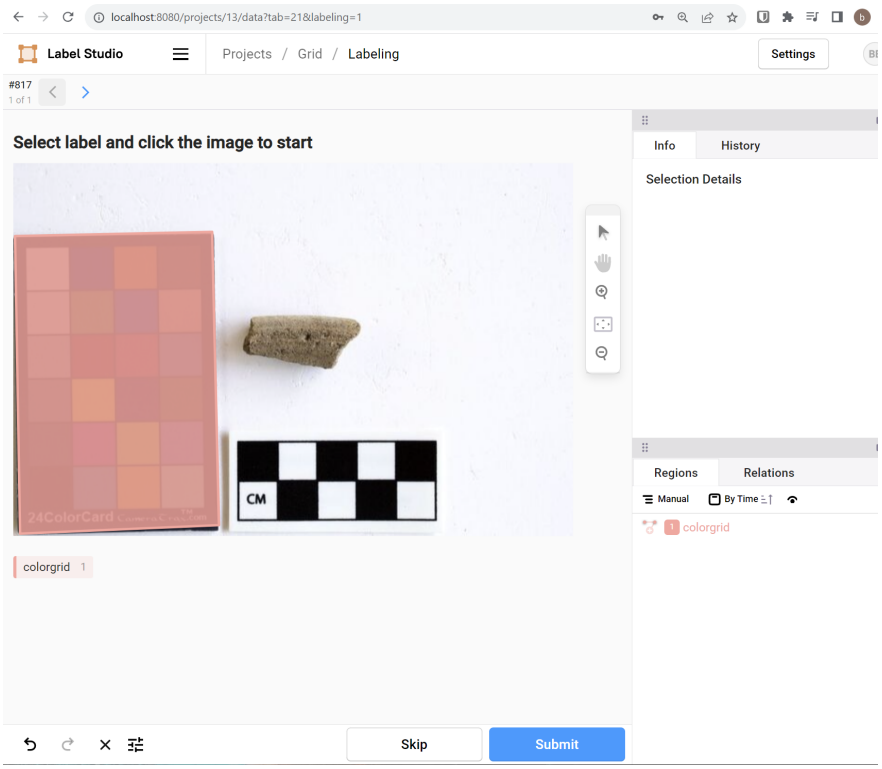
8. Click Save, then we can start label our images



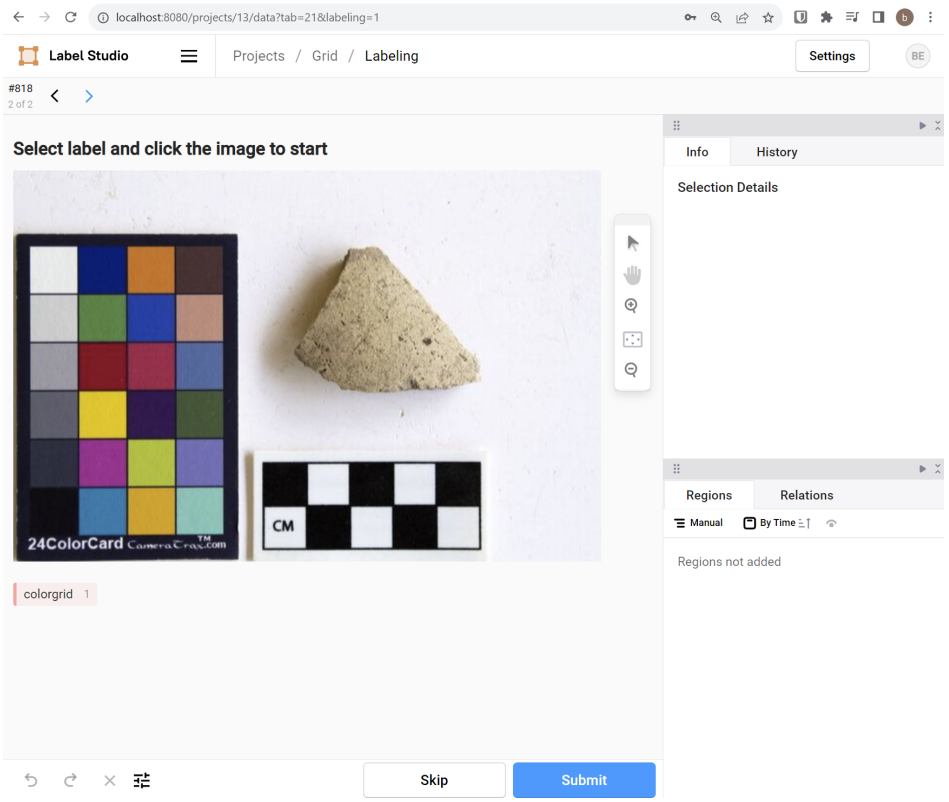
9. Click the blue button **Label All Tasks**, then you can see this page



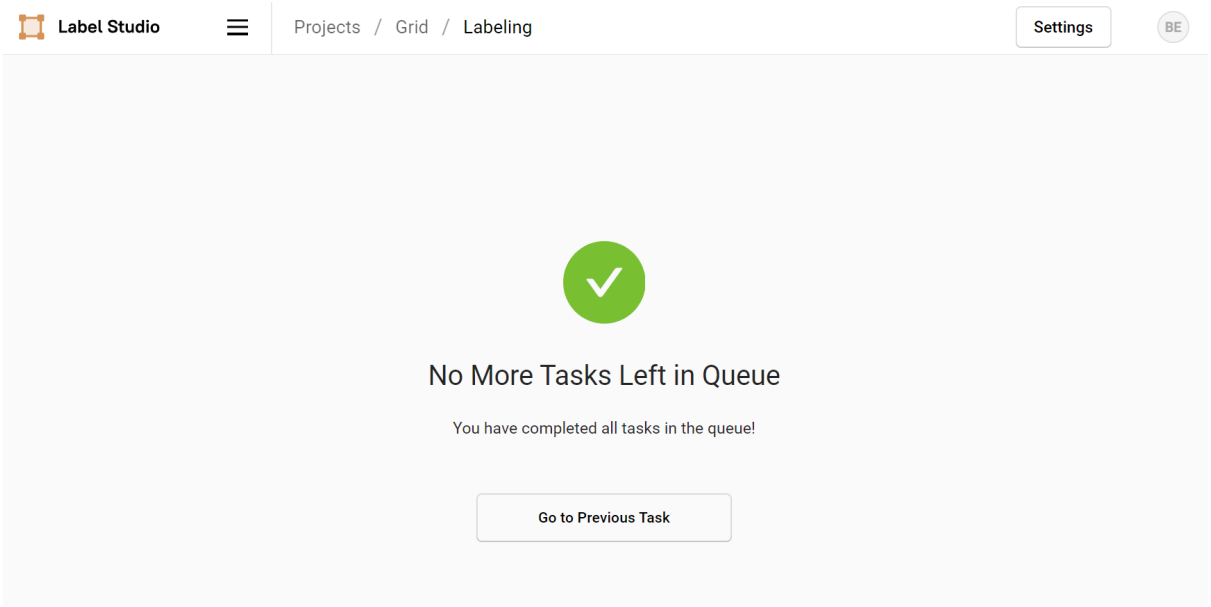
10. Click the **color grid button** in the left-bottom corner, then you can select points to enclose the grid in the image



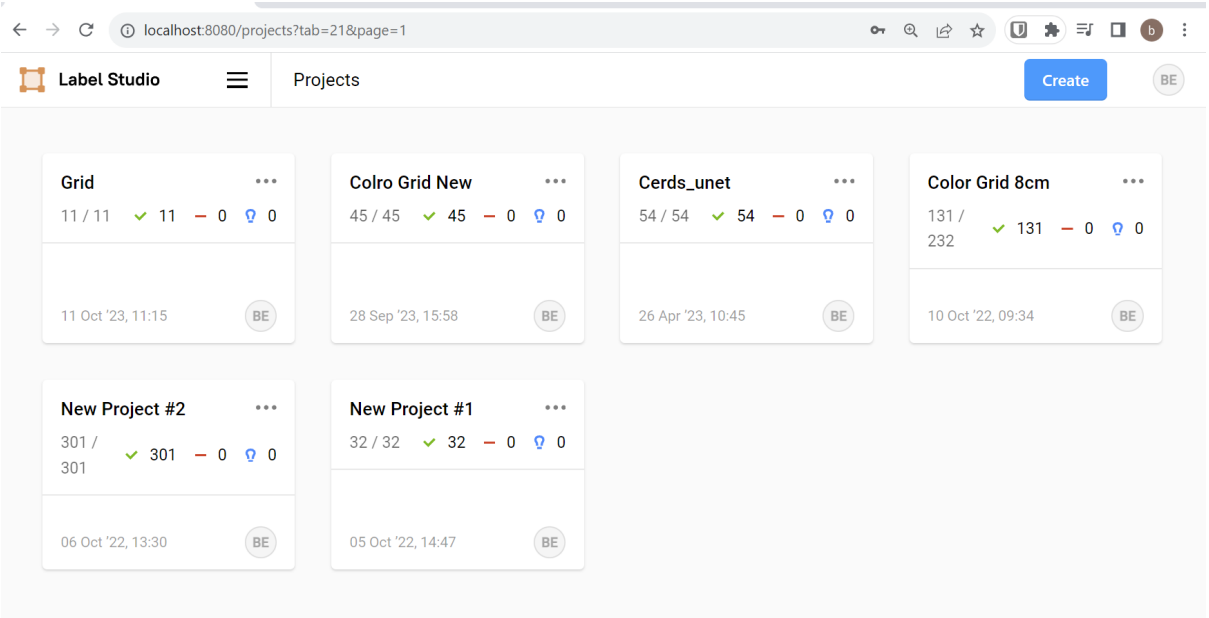
11. Click Submit, then you will label the next image



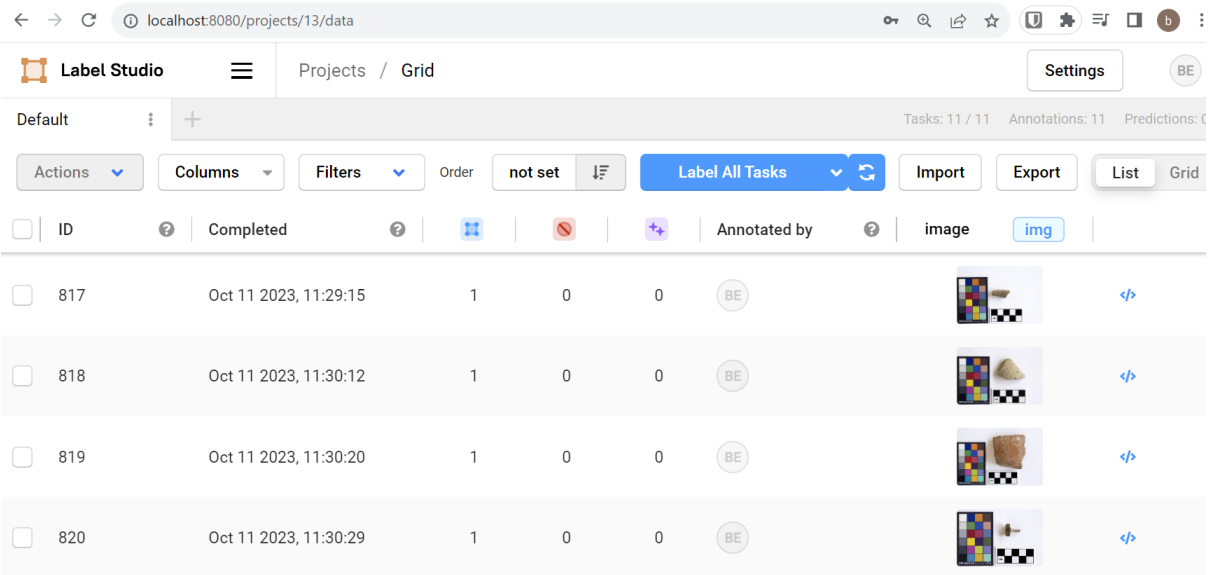
12. This page will show up when you finish labeling all images



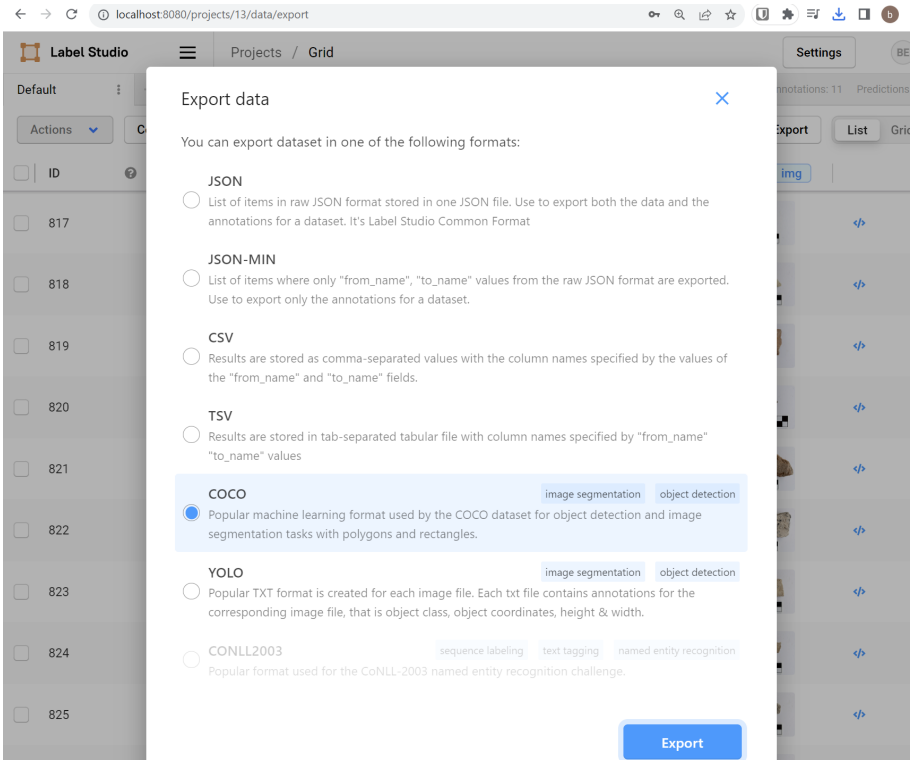
13. Go back to the projects page and choose the project you just finished labeling, which is **grid**.



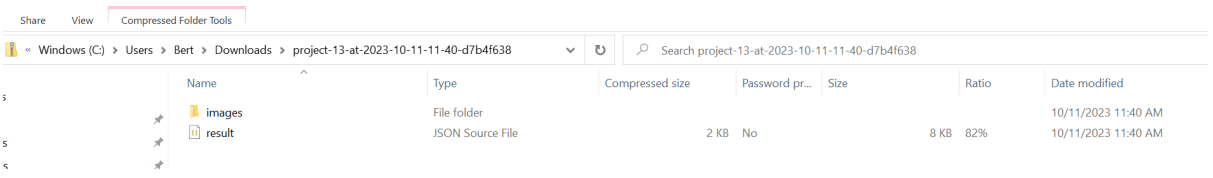
14. Click the **Export** button on this page,which is on the row of the blue button **Label All Tasks**



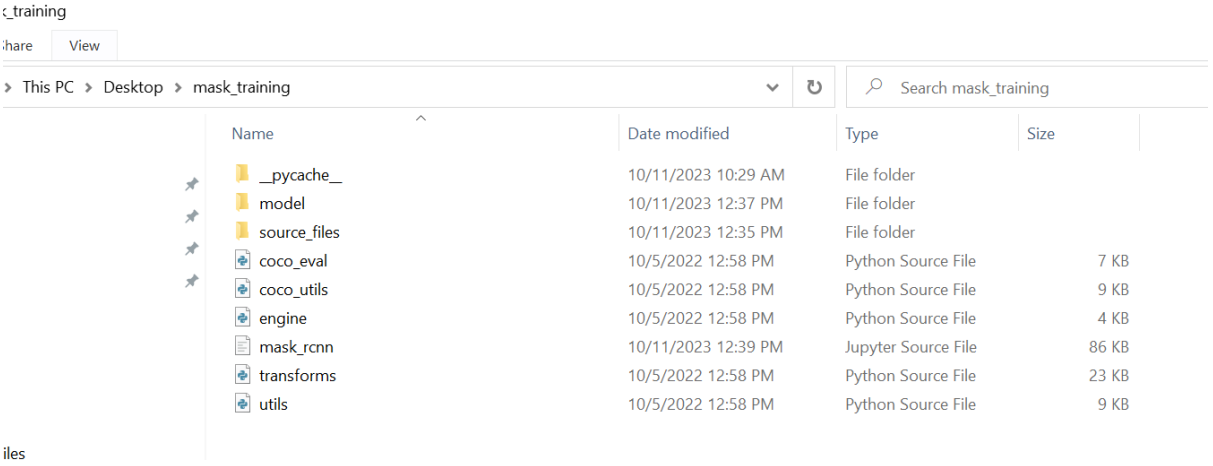
15. Choose the COCO option and click export



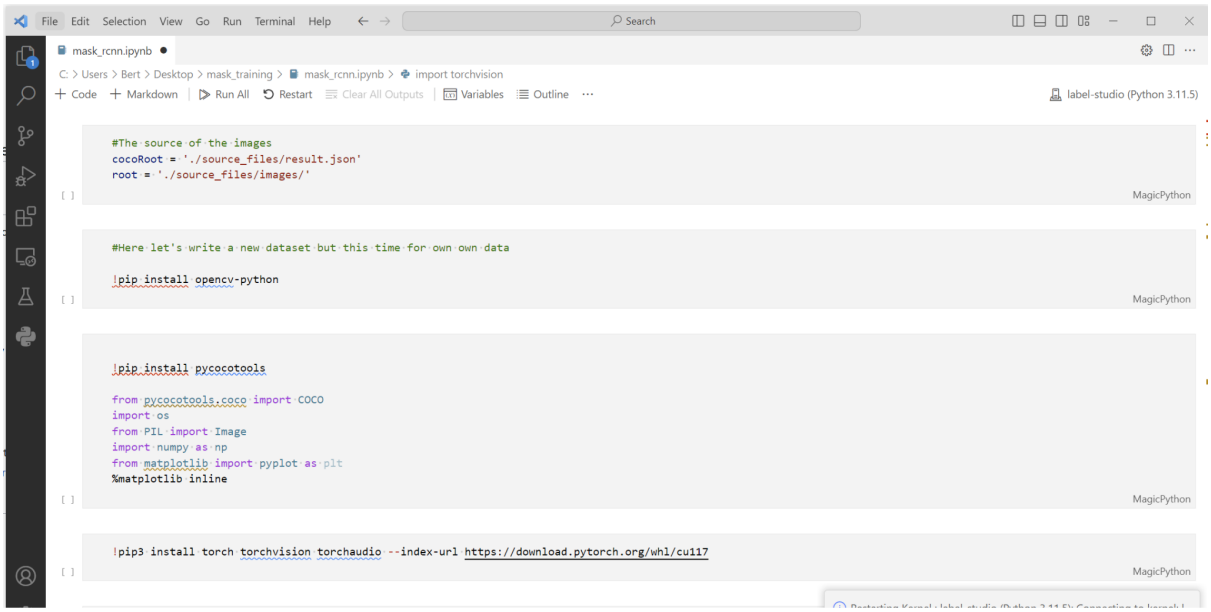
16. A zip file will be downloaded, inside of which we will have a **images** folder and a **result.json** file



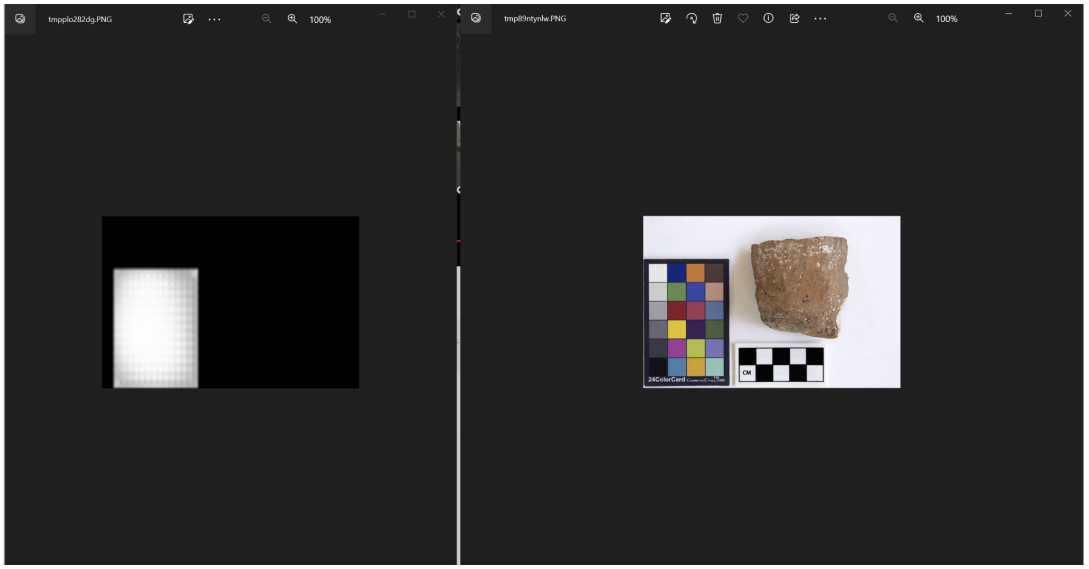
17. Copy them to the folder **mask_training's** subfolder **source_files.** The mask_training folder is in a zip in the location **E:\tools\APSAP\mask_training.zip**. Unzip the folder to a location in your user space.



18. Now open the **mask_rcnn.ipyb.** Choose the **label-studio** Conda environment we just created. Then press **Run All**. Libraries will be installed. (In rare cases, it may crash on the first run. Then you will have to click **restart** and **Run All).**



19. After the script ends running, it will open two images to indicate that the training has ended. This indicates the neural network has been trained



20. In the model folder, you can see the **train_model{time_string}.pt** here. The model has been trained!