Code Quality Review Report

Copy-Waste

Nolan Flegel, Rishabh Prasad, William Peers

Published: April 6, 2022

Table of Contents

1. Code Formatting

Alignments, formatting, whitespace

Our code uses Prettier and Python Intellisense In order to format our Typescript and Python code respectfully. Both of these are supported by the two main IDE's used by the team, Visual Studio Code and PyCharm. Intellisense offers a variety of features such as code completion, quick hover info, and quick fixes. Prettier automatically formats our TypeScript and JSON files, allowing us to have consistent styling across all files without needing to agree upon style conventions.

Naming Conventions

All of our Python code follows the PEP 8 Style Guide for Python Code. All local variable names as well as function names use lower case names with underscores in the place of spaces, otherwise known as snake case, and all class names use camel casing. Variables are named with a higher priority given to clarity and readability as opposed to abbreviating for the sake of short variable names. In any class method, self will be the first argument.

Code should have a standardised width

For better readability, we made use of the automatic line length limits in PyCharm and Visual Studio Code. Our code has a line length limit of 80 characters.

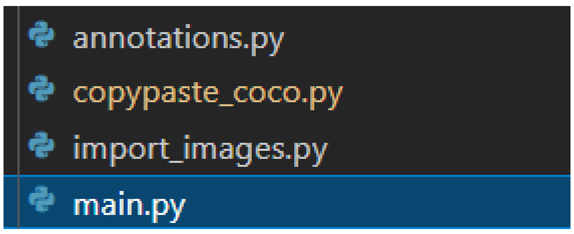
Removing Commented Code and Print Statements

Any code that is commented and not being used is removed before pull requests as well as print statements which were used for debugging. Finding these is part of the job of those reviewing each pull request.

2. Architecture

Separation of Concerns

We addressed separation in our codebase by following object oriented design. Each of our deliverables within our project has code separated into classes and functions which are imported when required. Within our copy-paste data augmentation codebase, we separate classes into different files as seen in the figure 1 below. Similarly, the code base for the front-end is split to group code which share similar functionalities such as pages, assets, charts, map and forms.



```
annotations.py
copypaste_coco.py
import_images.py
main.py
```

Figure 1: File Separation

Code Patterns and Technologies

Our codebase is written using popular and highly maintained programming languages, frameworks, and technologies. The code for the bin detector and the copy-paste pipeline is written in Python 3.9.5. It was chosen as Python has a large community of developers who build and maintain libraries which simplify our implementations. Our python projects are extended to achieve our goals with libraries such as opencv, numpy, torchvision, shapely, and pytest among many others. On the other hand, the dashboard uses the React framework with TypeScript, which is now considered the industry standard when building user interfaces. It also takes advantage of several popular libraries such as MapboxGL for the map and Material UI for commonly used components such as buttons and tables.

Design Patterns

The Model View Controller pattern is adopted for our data augmentation code base. As seen in figure 2 below, the user triggers the controller to begin an augmentation job. Then the controller requests images from our data storage to manipulate them. As the augmentation runs, the user is updated on the progress.

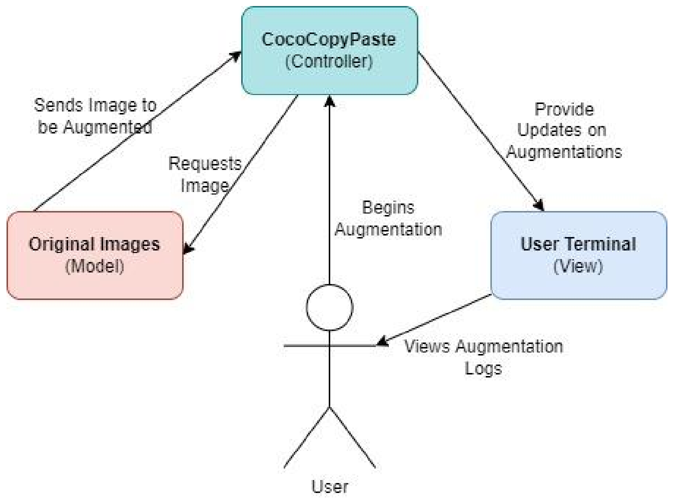
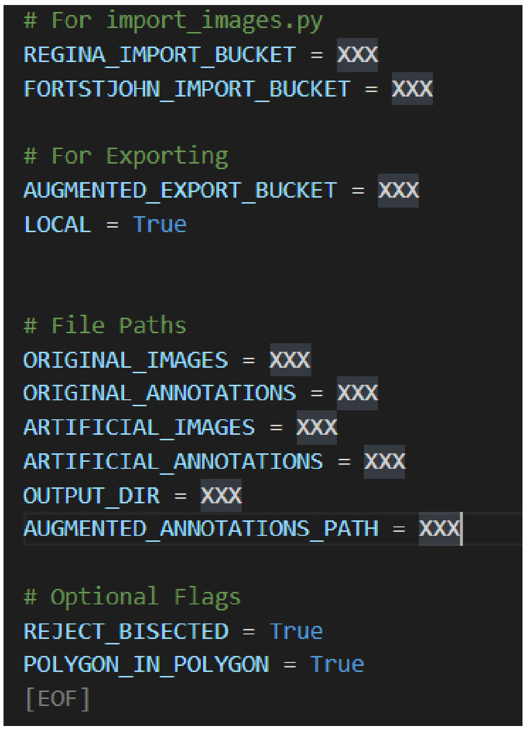


Figure 2: MVC Diagram for the Copy Paste Pipeline

3. Code best practices

No hard coding, use constants/configuration values

Any values which may be sensitive or need to change often are stored in an environment file instead of being exposed in our code. This includes file paths, S3 bucket names, flags, and file names. These environment files are either distributed amongst developers or created on their own so that these values are never uploaded to a repository. An example of this can be seen in figure 3 below.



```
# For import_images.py
REGINA_IMPORT_BUCKET = XXX
FORTSTJOHN_IMPORT_BUCKET = XXX

# For Exporting
AUGMENTED_EXPORT_BUCKET = XXX
LOCAL = True


# File Paths
ORIGINAL_IMAGES = XXX
ORIGINAL_ANNOTATIONS = XXX
ARTIFICIAL_IMAGES = XXX
ARTIFICIAL_ANNOTATIONS = XXX
OUTPUT_DIR = XXX
AUGMENTED_ANNOTATIONS_PATH = XXX

# Optional Flags
REJECT_BISECTED = True
POLYGON_IN_POLYGON = True
[EOF]
```

Figure 3: Environment Variables

Removing Unnecessary Comments

One of our highest priorities when writing code is readability. Because of this, if we feel a need to explain our code with a comment we are inclined to attempt to rewrite the code to be easier to understand instead. Comments are only made to explain why something is happening, not what is happening.

Avoid Multiple if/else Blocks

Our code avoids heavily nested if/else blocks as these are confusing to trace through. There are only a few instances where we have if/else blocks that are nested once as sometimes it is necessary.

Use Library Features When Possible

There are many libraries which are used throughout our Python code, some of the most prominent are numpy, cv2, shapely, and coco. These all provide useful classes and methods for handling images, segmentation masks, polygon masks, and calculations involving those. Using these libraries allows us to allocate more time for creating something new. An example of this can be seen in figure 4 below.

```
from shapely.geometry import Polygon
from datetime import datetime
from pycocotools import mask as maskUtils
from utils.config import load_config
```

```python
from annotations import build_image_json
import boto3
from pathlib import Path
import numpy as np
import albumentations as A
import cv2
import os
```

Figure 4: Libraries Used

4. Non Functional Requirements

Maintainability

We used standardized conventions in our workflow and our code to improve readability. We added a commit message template to our GitHub repository Readme that users can follow when committing code to our GitHub repository. This keeps messages standardized and easy to interpret. The convention we used can be seen in figure 5 below.

```
All commits should follow the same schema:
- git commit -m task(scope): brief description of work

Where Tasks is one of:
- feat: A New Feature
- fix: A Bug Fix
- docs: Documentation Only Changes
- style: Changes that don't affect the meaning of code (white-space, formatting, ect)
- refactor: A code change that neither fixes a bug or adds a feature
- perf: A code change that improves performance
- test: Adding tests
- ci/cd: Changes to the build pipeline
- chore: Changes to auxilary tools and libraries such as documentation generation
- config: Changes to project configuration
```

Figure 5: Git Commit Conventions

Another standard used in this project is the PEP 8 Style Guide for Python Code. This helped make our Python code consistent and readable. We were able to use minimal comments to explain functionality as the function and variable names were descriptive and consistent in keeping with this standard.

The configuration and logging of our application uses python libraries to consolidate and simplify system management. We use environment files to obscure sensitive information and provide a single location for configuring system variables. Error and status metrics are logged with descriptive messages to a system log file which is stored with each data augmentation run.

Reusability

Throughout our codebase we developed functions to be reusable by following the Don’t Repeat Yourself principle. In order to maintain this principle we extracted code which was repeated and created simple, single responsibility functions which could be called instead. An example of this is the “get_objects” function within CocoCopyPaste, where objects for both original and artificial images can be extracted. Although there are minor differences in how they are extracted, we account for the differences by passing a simple “is_orginal” boolean parameter. When an original image is sent to this function, a small block of code which makes the artificial objects different is skipped. On the front-end, this is primarily implemented by extracting components which are used multiple times such as buttons, charts, and tables.

Reliability

Our copy paste data augmentation code is built to be robust as we catch all errors and exceptions. We log these errors to be able to identify points of failure within our code and are able to fix them in future iterations. Catching these errors is also beneficial as an augmentation job is never completely cancelled when an error occurs. Additionally we utilize pytests to extensively test our code as new features are implemented. The dashboard is also tested using unit tests developed with Jest to verify any calculations which are made for information displayed on the front-end.

Extensibility

Our methods are encapsulated in Object Oriented classes and can be easily extended with new features. Functions are designed to be single purpose and loosely coupled, this allows for new features to be added without inherited dependencies. We extend a number of classes throughout our data augmentation pipeline to allow compatibility between different tasks.

Security

The security of our users is handled by AWS Cognito and the sign-in process was developed by the Prairie Robotics team. Cognito ensures that the user’s login information is authenticated through an encrypted process. This is significant as Cognito handles the complex process of guaranteeing security, which is a difficult task to achieve for us as a small team.

Usability

From a developer’s perspective, we designed our code to use popular standards and conventions so that any new developer can adopt our project and extend it. Also from the user’s perspective, our front-end is designed with familiar knowledge for waste management workers and with a limited number of controls. This makes our product intuitive and an effective solution for new users to adopt as the learning curve is minimal.

Performance

Processing data and executing machine learning algorithms is a resource intensive endeavour. Datasets are often tens or hundreds of gigabytes in size and machine learning training frameworks operate best on systems with increased ram and dedicated graphics processing units. Cloud computing is an ideal platform for machine learning because of the dynamically scalable resources. A comparison of the runtime of a machine learning training job revealed that a laptop or desktop computer could complete the task between 4 and 12 hours while an Amazon Sagemaker instance could complete the same task in 35 minutes.

Our data augmentation pipeline incorporates both cloud services and a local computer. This is not ideal and a future goal is to implement this system entirely within cloud services. A particular bottleneck in performance is uploading the augmented images to cloud storage. As images are created, they are uploaded to an Amazon S3 storage bucket. The machine running the augmentation script must create a temporary S3 client each time it uploads an image and wait for a response before starting the next augmentation. This can become a significant problem for datasets of sufficient size.

Scalability

The design of our data augmentation pipeline allows it to run many instances in parallel. The system is limited by the number of cloud resources you have available. This is useful for the experimentation and customization of different datasets and allows a user to quickly test and compare different augmentation parameters.

In much the same way, our machine learning scripts are designed to run in parallel using Amazon Sagemaker. Each job is run on a separate instance and stored with a unique ID when completed. Training jobs can run simultaneously, with different parameters and have no impact on individual completion performance. A user is only limited by the number of cloud resources they have provisioned.

5. Object Oriented Design Principles

Following the Single Responsibility principle we simplified functions down to single tasks. We separated many common functions into separate files where they could be imported separately and reused repeatedly. The open / closed principle is achieved in our code through the use of classes within our data augmentation codebase. These classes are limited to only include functions which are required for the purpose of the class. but can be extended when new functions are implemented.

Interface segregation does not exactly apply to our deliverables, however, interfaces were used when passing objects on the front-end. Providing these interfaces within each function ensures objects have accurate values associated with them. We follow the Liskov Substitution Principle when we extend the COCO object in our augmentation pipeline. We are able to replace the Superclass instance with our own object and methods while utilizing the built-in methods without breaking functionality. Dependency injection is a principle that we found difficult to implement. With our image transformations, we identified some issues during debugging that are a result of not abstracting lower level functions properly. This is a principle that we need to spend more time practicing and look for ways in which we can improve.