Final Report

Team Deinonychus – Patrick Creighton, Luka Rogic, Christopher Lee, Juntong Luo

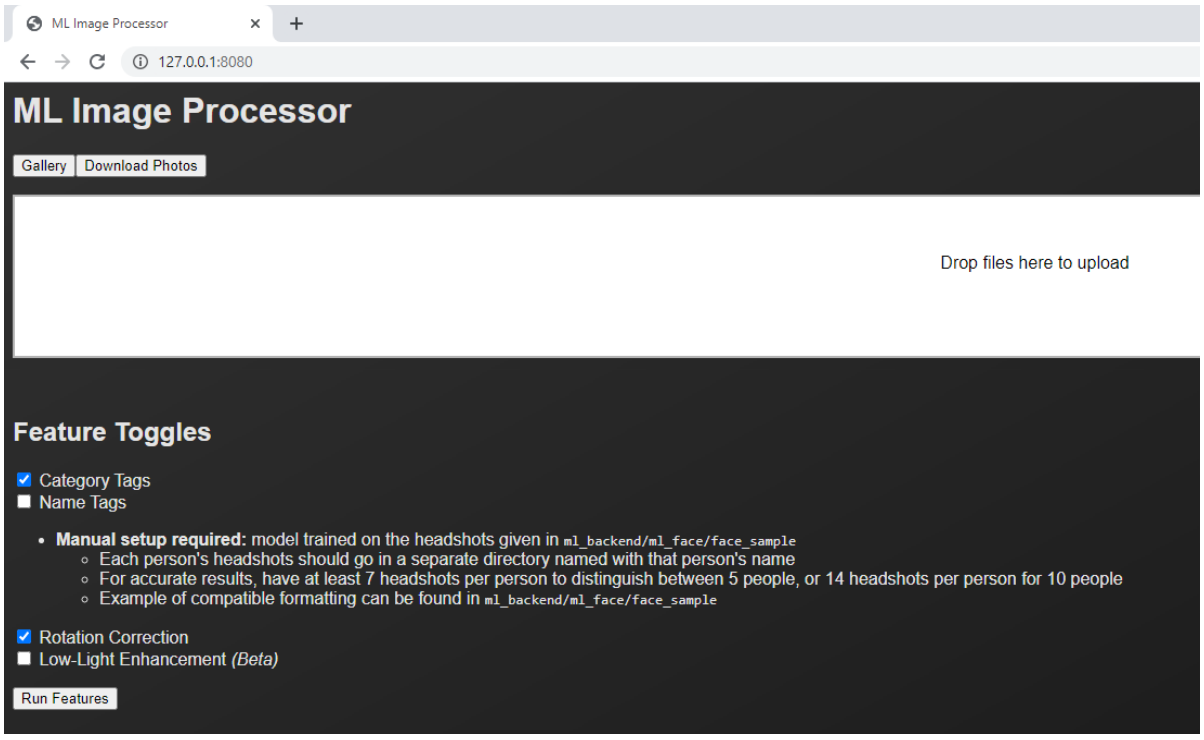
Abstract

This application aims to use machine learning to simplify and speed up the image organization process. First, a user of this application would upload images that they want organized. These images will then be run through multiple ML models. The application will first identify objects (i.e., person, tree) in the images, which would be used to assign them category tags (i.e., portrait, nature). Other supported features include facial recognition, rotation correction, and low-light enhancement. The user can then look through the images, manually editing the tags of those that were classified incorrectly. When they are finished, they can download the tagged and enhanced images.

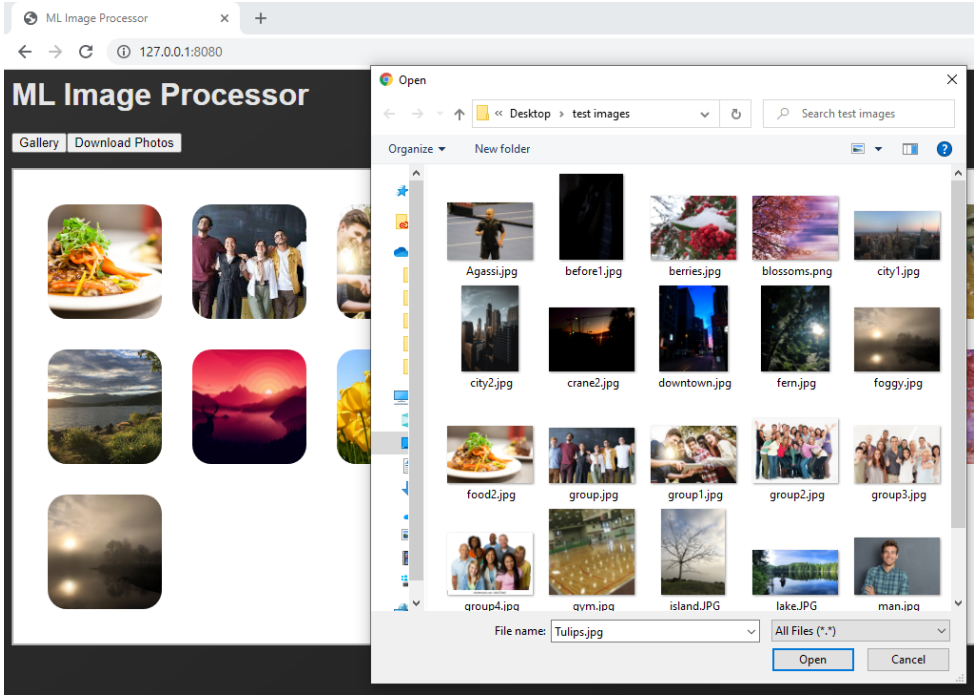
Application Walkthrough

1. Upon opening the application, a user would be greeted by the homepage

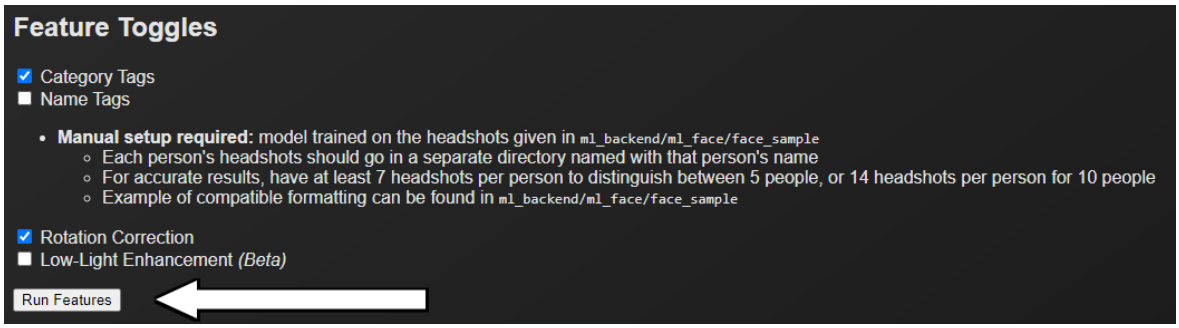
- Homepage also contains application usage instructions



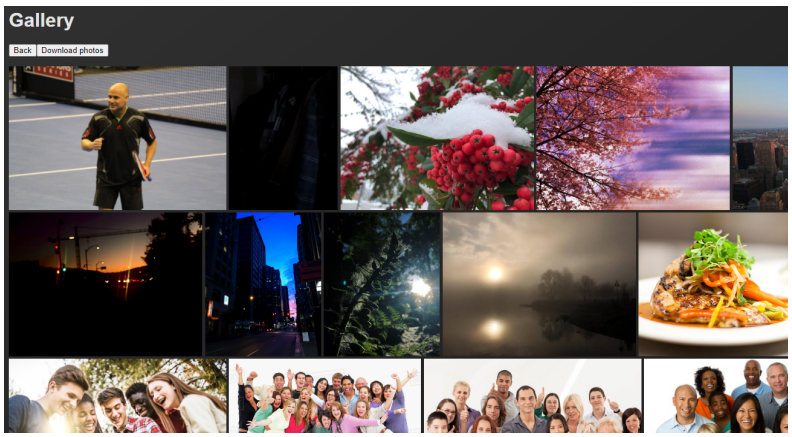
2. Next, they upload images from their computer
    - By clicking on the white box to select which images, or dragging them to the box



3. The user selects which machine learning models they would like to enable
    - Clicking the **Run Features** button will run the images through the selected models

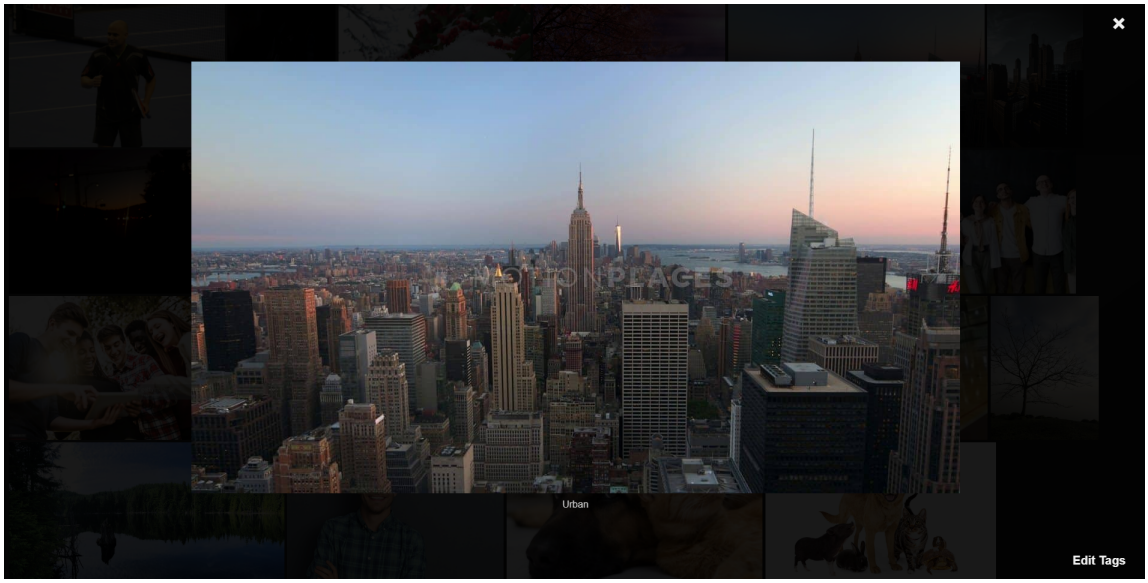


4. Clicking the **Gallery** button will open the gallery page



- Uploaded images are displayed with their original aspect ratios

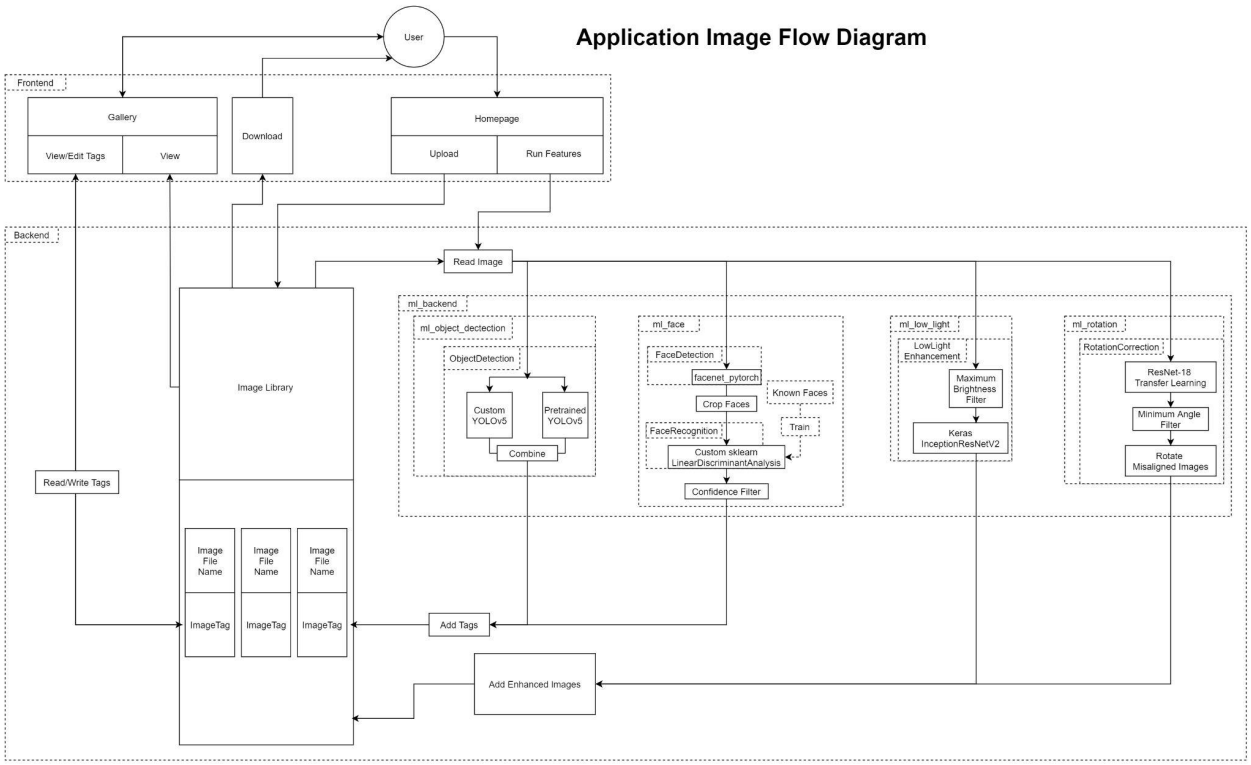
5. Clicking on an image will expand it and display their tags
    - Clicking the **Edit Tags** (bottom-right) button allows the user to manually edit an image's tags



6. Clicking the **Download Photos** button will download the tagged images
    - If the rotation correction and low light enhancement features were selected, their output photos will also be in the downloaded zip folder

Name	Date modified	Type	Size	Tags
correct_rotation	2021-05-16 8:20 PM	File folder		
Agassi.jpg	2021-05-16 8:23 PM	JPG File	83 KB	Portrait; Sports
before1.jpg	2021-05-16 8:23 PM	JPG File	148 KB	
berries.jpg	2021-05-16 8:23 PM	JPG File	149 KB	Nature
blossoms.jpg	2021-05-16 8:23 PM	JPG File	431 KB	Nature
city1.jpg	2021-05-16 8:23 PM	JPG File	70 KB	Urban
city2.jpg	2021-05-16 8:23 PM	JPG File	48 KB	Urban
crane2.jpg	2021-05-16 8:23 PM	JPG File	43 KB	Urban; Nature
downtown.jpg	2021-05-16 8:23 PM	JPG File	69 KB	Portrait; Urban
fern.jpg	2021-05-16 8:23 PM	JPG File	116 KB	Nature
foggy.jpg	2021-05-16 8:23 PM	JPG File	48 KB	Nature
food2.jpg	2021-05-16 8:23 PM	JPG File	65 KB	Food
group.jpg	2021-05-16 8:23 PM	JPG File	12 KB	Group Photo
group1.jpg	2021-05-16 8:23 PM	JPG File	114 KB	Group Photo
group2.jpg	2021-05-16 8:23 PM	JPG File	161 KB	Food; Group Photo
group3.jpg	2021-05-16 8:23 PM	JPG File	47 KB	Group Photo
group4.jpg	2021-05-16 8:23 PM	JPG File	35 KB	Group Photo
gym.jpg	2021-05-16 8:23 PM	JPG File	35 KB	Portrait
island.jpg	2021-05-16 8:23 PM	JPG File	153 KB	Nature
lake.jpg	2021-05-16 8:23 PM	JPG File	441 KB	Nature

Diagram of the Images' Flow Through the Application



This diagram depicts the application structure and flow in which images move through the application. The application is separated into two parts, frontend and backend. In the frontend, we have a homepage and gallery page, where the user interacts with the application. When the user uploads images on the homepage, the Image Library stores uploaded images and their tags in the backend. After uploading images, the user can click the "Run Features" button to run the models they selected, a set of the models can be found in the project's "ml_backend" directory. Then, the models will obtain images from the library, and return their outputs (either tagged or enhanced images), back to the library. There are four features in the backend:

- For object detection, images run through the custom and the pretrained YOLOv5s models separately, and their output tags are combined and added to the library.
- For face recognition, the facenet model crops faces out of the images, and the faces are fed into the scikit-learn SVM, which is trained on known face images. The SVM then outputs their predicted names, along with its confidence. After that, a filter is applied to drop names that the model has a low confidence in.
- For low light enhancement, the application checks the average brightness of the image and runs the model only on the dark ones.
- For rotation correction, the ResNet-18 model predicts the degree of misalignment in images, and the ones with an angle greater than a given threshold are straightened.

After the ML models are done, the user can go to the gallery page to view images and edit their tags, or download all tagged and enhanced images from the library by clicking the download button.

Project Components

Object Detection: YOLOv5 models

The pre-trained component of the YOLOv5:
This component, which is used to detect objects from the COCO dataset, was not implemented by us. We installed the weights file from the YOLOv5 repository and connected it to our web application to tag images. Since it was proven to be an accurate model, we decided to detect the additional objects that we would need with a separate YOLOv5 model.

The custom component of the YOLOv5:
This component was trained by us and used to detect objects from 4 classes, which we decided on by choosing representative objects of more general tags/categories (listed in the Tags and Objects PDF), and seeing which ones were not in the COCO dataset.

The main function we had to write to form the dataset was called DarknetFormat. Its input are two lists, for the names and quantity of images desired of each object/class (which had to be available on the Open Image Dataset website). It looks for images that contain one or more of the desired objects, then returns the image ID and bounding boxes for those identified objects. Lastly, this information is formatted to fit the YOLOv5 format.

Next, the output of DarknetFormat is used to set up the label text files and the input for the Open Images Downloader (which takes in a list of image IDs to download). After the images are downloaded, the images and labels are split into training and validation folders, and configuration files required by the YOLOv5 model are created. The last step is to run the YOLOv5 train command to train the model on our curated dataset.

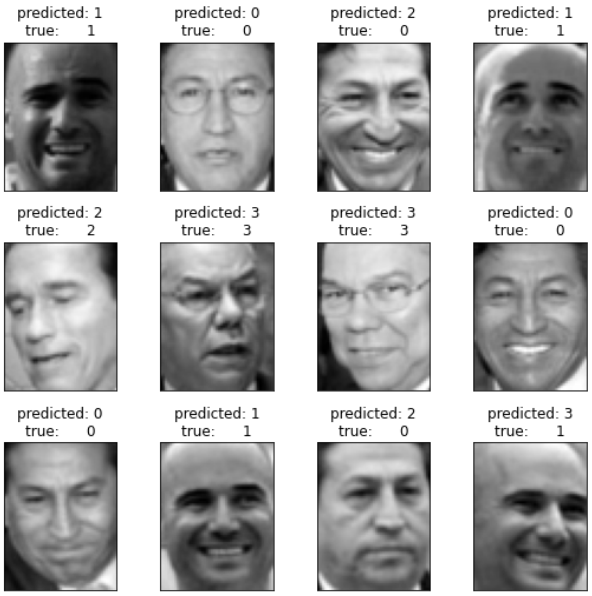
The resulting model performs quite well, particularly in detecting skyscrapers and plates (60-80% confidence). Despite this, it can sometimes struggle to find nice bounding boxes, and ends up putting many overlapping boxes for one object (see the tree and flower examples below). However, this does not matter for our use case, as we only need to detect the presence of certain objects in an image.

Face Detection: Facenet Pytorch MTCNN

The Facenet Pytorch is an MTCNN implementation for face detection. We use this model for finding the bounding box of faces in images. Then, the faces are cropped and fed into the face recognition model. We decided to use a proven, pretrained model since it serves as the input to another model, requiring it to be as accurate as possible.

Face Detection: SVM with LDA

The face recognition model was implemented with a support vector machine (SVM) from scikit-learn, using a Linear Discriminant Analysis (LDA) classifier. We decided to use a SVM because quick training time was a priority, as the model had to be trained on the user inputs. To facilitate the loading of custom datasets into Colab without user input, we pushed them to the "Curated_Datasets" directory of our repository, cloning that directory into our notebooks. These datasets were compressed to .tar.gz files, to increase performance and save space. To determine the optimal model to use, we thoroughly analyzed the performance of two models over two different datasets, Olivetti and Labelled Faces in the Wild. The conclusions drawn can be found in `ML_Development/Face Recognition Analysis.pdf`. For our model to achieve 60-80% accuracy, 7 images per person in the training dataset would be required to distinguish between 5 people, or 14 images per person to distinguish between 10 people. To be able to train on faces selected by users within the application, we adapted code from scikit-learn's code base to form datasets from image folders. However, we were not able to fully implement this feature for the project deadline, and currently the image folders need to be set up manually. See the Milestone 4 Report, Current Challenges / Next Steps 2 for more details.



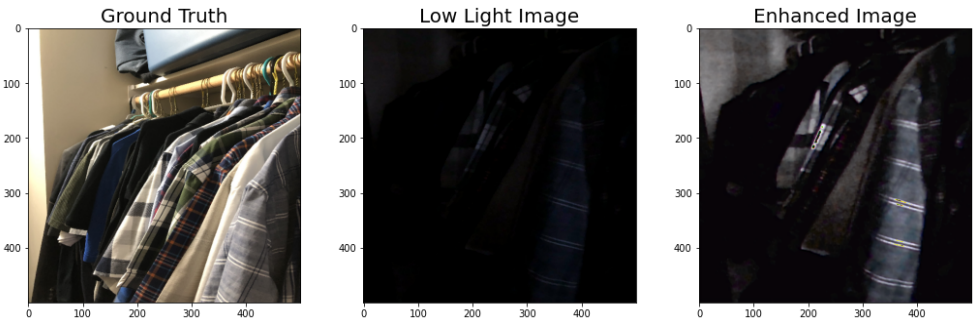
Rotation Correction: ResNet-18 model

The rotation correction model was implemented in PyTorch using a ResNet-18 model with transfer learning. This was chosen because of its ease of use and our familiarity with these tools. The dataset was formed by taking images from the Google Scraped Image Dataset, then rotating and cropping them so that there are no black bars. In order for the images to work with code from past assignments, we had to resize and convert the NumPy / OpenCV 2 image to a 224x224 PIL image. The label of an image is how much it was rotated by, which was a randomly selected value between -15° and 15°. The L1 loss function was used to train the ResNet-18 model to predict values as close to the labels as possible. Training on this dataset, our model managed to predict how much an image was misaligned to within a couple degrees, achieving greater accuracy when there were well defined vertical and horizontal lines. If we were to continue application development, we would look into adding the ability to manually adjust image rotation.

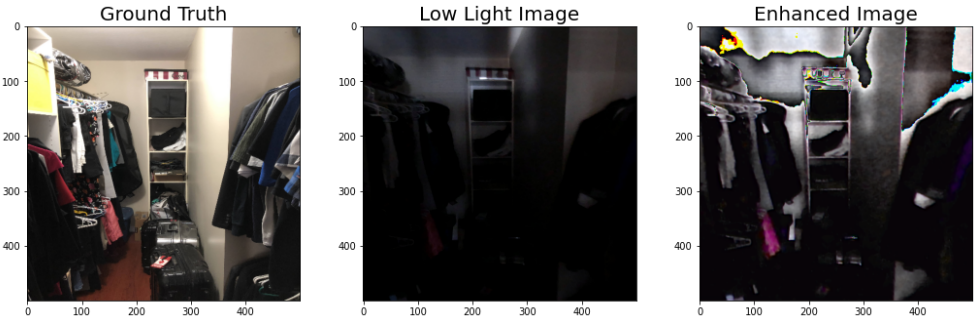
Low-Light Enhancement: Keras InceptionResNetV20

The low-light enhancement model was implemented using a Keras Pretrained InceptionResNetV2 model with transfer learning, adapted from a past Kaggle project. The dataset was formed by taking images from the Google Scraped Image Dataset, adding grain (which is often found in low-light images) and decreasing their brightness. The label of an image was the unmodified image. For images where we artificially added grain and decreased brightness, the model was able to achieve remarkable results. Real-world performance, however, was more hit or miss. If an image was too dark, the model would not be able to detect any color in the image to enhance. If there were bright spots in the image, the model could produce neon spots. This feature is still in beta because these spots ruin an image. Moving forward, we might have to rethink how to implement this feature in order to avoid the issue. A temporary fix would be to finetune our filter to reject images that are too dark, or have a high contrast.

Good Example



Bad Example



Web Application: Flask

Our web application was developed using Flask, which was chosen due to the vast amount of tutorials and resources available online, as well as its ease of use. Accompanying Flask, which structured the webpage's backend, the HTML, CSS, and JavaScript languages were used in the frontend to design the appearance and user experience of our application. Initially, the application consisted of a simple image uploader. As the weeks progressed, so did our application, with the addition of features that included downloading uploaded images, the gallery page, and most importantly, integration with our machine learning models. Once we were satisfied with the performance of these core features, we continued to add additional improvements to the user experience, such as the ability to manually edit tags and a loading screen that displays when the machine learning models are running. A recurring challenge with the web application was running it on the Google App Engine platform. Our original plan for the project was to host it on the Google App Engine, allowing the public to try out our work without having to install our project files and required libraries. However, with every new feature we added, a new issue would come along with it. For example, when we first added the download feature, the application would only work locally because the Google App Engine only allows certain directories to be modified. We eventually overcame this particular problem, but due to the frequency of these types of problems appearing, our team decided to focus our efforts developing the application's core features when run locally. If we were to continue with this application's development, deploying it on Google App Engine for public use would be one of our top priorities.

Image Library: pyexiv2

We developed a class named Image Library to manage image files and tags. Other parts of the application interact with Image Library to read/write tags. When the user clicks the download button on the webpage, the library uses the pyexiv2 package to store tags into the image files by modifying their metadata. For storing tags, three types of metadata work, but we chose to use "Xmp.dc.subject" because it has the best compatibility across different operating systems.

Team Member Contributions

Christopher

- Developed frontend of application using Flask and its core features including the image uploader/downloader and the gallery page
- Ensured the application was compatible with the Google App Engine for deployment before the group decision to focus on developing the application's core features when run locally
- Added a PNG to JPG converter to the image uploader since our ML models are only compatible with JPG images
- [With Juntong] Added code to display tags on the webpage
- Added features to improve user experience such as the ability for user to manually update tags on the webpage's gallery, and a loading page for when the ML models are processing photos

Luka

- Researched and helped decide on the machine learning model for the object detection.
- Explored using different python scripts for downloading images from Google's Open Images Dataset.
- Worked with Open Images website to download and format images/labels according to Yolov5 format for proper training (see the Open Images Downloader Finished notebook).
- [With Patrick] Added code that maps objects detected in the image to their category tags.
- Trained the model on classes from the Tags and Objects PDF with different parameters to find the best accuracy weights file.

Juntong

- Looked into how tagging works for images, tested the compatibility of different metadata for storing tags on Windows and iOS, and saved the findings in "tagger/Metadata for Tags.pdf".
- Developed the tagger package to manage tags of images and support reading/editing/storing the tags.
- Developed the ObjectDetection class to run the custom and the pre-trained YOLOv5s models to find tags of images.
- Researched about the facenet_pytorch model and utilized it to detect and crop faces out of images.
- Applied the rotation correction model to automatically correct tilted images.
- Connected different ML models to the application.
- [With Christopher] Added code to display tags on the webpage.
- [With Patrick] Implemented feature toggling on the homepage.
- Made the "Diagram of the Images' Flow Through the Application".

Patrick

- Object detection model development
    - Wrote a function to return images with at least one object for any of the objects in a list
    - [With Luka] Added code that maps objects detected in the image to their category tags
- Face recognition model development
    - Conducted a detailed analysis of the performance of two face recognition models, as well as which classifier to use
    - Added confidence levels to the model
    - Made custom datasets wrote code to load them into Colab without user input
    - Adapted code to form datasets from image folders
- Rotation correction model development
    - Adapted code to return the rectangle with the maximum area given the degrees by which it is rotated
    - Adapted code to visualize model's performance
- Low-light enhancement model development
    - Took photos to use as real-world performance metrics, and analyzed results
- Frontend
    - Wrote application usage instructions and tweaked the design of the homepage
    - [With Juntong] Implemented feature toggling on the homepage
- Repository organization, cleanup, and documentation
    - Goal is to enable other programmers to run and build on our application

Conclusion

Reflecting on our experience developing ML Image Processor, there were several key takeaways that we had. First, group communication and organization are important, especially without the option to meet up in person. This led us to adopt several strategies to work together effectively, including recording meeting notes and pair programming. In the beginning stages of the project, we mostly worked on separate tasks. When doing so, we strived to write clear and well-documented code, proving useful in later stages where we had to integrate the individual components into one cohesive application.

Regarding the web development side of the project, we were introduced to the HTML, JavaScript, and CSS languages, and further increased our experience with Python. Our application was developed using Flask, which was another useful tool that we now have under our belt. Regarding the machine learning side, we built on existing models by training them on our own custom datasets, so that they can be used to meet our needs. In particular, creating and formatting high quality datasets was essential to the success of our models. In conclusion, being able to apply our machine learning knowledge that we developed over the term, and see our project come to life and fulfill the goals set out in the proposal, was a rewarding experience in which we learned many transferable skills.