Documentation
Technical Test

By Izzan Silmi Aziz

Table of Content

1. Objective

The objective is to develop a simple model that can detect whether someone is wearing a mask or not.

2. Model

In this project, the model used to complete this is using haarcascade_frontalface_default.xml to detect the position of the face and haarcascade_mcs_mouth.xml to detect the position of the mouth. When the two models are combined, it can detect whether someone is wearing a mask or not. Since the technique developed by Paul Viola and Michael Jones in 2001, Haar features and Haar cascades have revolutionized object detection. They have become integral components in various applications, ranging from facial recognition to real-time object detection.

Haar features are extracted from rectangular areas in an image. The feature's value is based on the pixel intensities. Usually, it is calculated using a sliding window, and the area within the window is partitioned into two or more rectangular areas. Haar feature is the difference in the sum of pixel intensities between these areas.

It is believed that an object's presence will distort the variation of pixel intensity. For example, the background is usually in a uniform pattern, in which a foreground object will not fit. By checking the pixel intensity between neighboring rectangular areas, you should be able to notice a difference. Hence it is indicative of the object's presence. Haar cascade combines multiple Haar features in a hierarchy to build a classifier. Instead of analyzing the entire image with each Haar feature, cascades break down the detection process into stages, each consisting of a set of features. In OpenCV, there are pre-trained Haar cascade classifiers. The pre-trained classifier is stored as an XML file. So in this project, OpenCV optimizes the model.

Following are the steps for creating a model in code (**model.py**):

a. Import necessary libraries: These are the OpenCV (cv2) and NumPy libraries, commonly used for image processing tasks.
b. Define the function detect_mask: This function takes two parameters - namefile (the name of the output file) and file (the image file).
c. Set up parameters and constants: These variables include font type, text size, font colors, thickness of the text, font scale, and threshold for binary conversion.
d. Read the image: This line reads the image using OpenCV, converting it from a byte stream to a NumPy array.
e. Convert the image to grayscale: Converts the image to grayscale for easier face detection.
f. Apply thresholding to create a binary image: This step helps in detecting faces more effectively.
g. Detect faces in both the grayscale and binary images: Cascade classifiers are used to detect faces in both the grayscale and binary images.
h. Handle cases where no face is found: If no faces are detected in both images, a message is added to the image, and the condition is set accordingly.
i. Handle cases where only one face is found in the binary image: If only one face is found in the binary image, it assumes a masked face.

j. Process cases where faces are found in the grayscale image: For each detected face, a rectangle is drawn around it, and a region of interest (ROI) is extracted in grayscale.
k. Detect mouth rectangles within the face region: Using a mouth cascade classifier, it attempts to detect mouths within the face region.
l. Check if no mouth is detected: If no mouth is detected, it assumes a masked face.
m. Process cases where mouths are detected: If mouths are detected within the face region, it assumes a face without a mask and updates the condition accordingly.
n. Save the annotated image: The final annotated image is saved with the specified filename.
o. Return the condition: The function returns the condition, which represents whether a face is found, has a mask, or doesn't have a mask.

3. Inference

After creating the model function, it is necessary to test the model function with code (**inference.py**):

a. Import the detect_mask function from the model module: the detect_mask function is defined in a module named model, this line imports the function for use.
b. Define the inference function: This function takes two parameters - input_path (the path to the input image file) and output_path (the path to save the output image with detections).
c. Open the input image file in binary mode: This uses a with statement to open the input file in binary mode ('rb'). The file is automatically closed when the block is exited.
d. Call the detect_mask function to perform inference: The detect_mask function is called with the output path and the opened binary file. The result is stored in the result variable.
e. Print the result of the inference: The result of the inference (whether a face is found, has a mask, or doesn't have a mask) is printed to the console.
f. Check if the script is being run as the main module: This condition ensures that the following code block is only executed when the script is run directly, not when it's imported as a module.
g. Set up an argument parser: An ArgumentParser object is created to handle command-line arguments. Two arguments are defined - input for the input image file path and output for the output image file path.
h. Parse the command-line arguments: The parse_args() method parses the command-line arguments and stores them in the args variable.
i. Call the inference function with the provided input and output paths: The inference function is called with the input and output paths obtained from the command-line arguments.

	
Input image	Output image

4. Deployment

After the model function is running well, an API is created, which can later be deployed on a remote server. The following are the steps carried out:

a. Build router for API:

I. Creating a function for process image with model (**mask_inference.py**):

- Import necessary modules: These are the libraries and modules used in the code, including time for measuring processing time, OpenCV (cv2) for image processing, lru_cache for caching, os for file path operations, numpy for numerical operations, and PIL for working with images.
- Decorate the process function with LRU caching: The @lru_cache decorator is used to cache the results of the function. It helps in memoizing the function's output to avoid redundant computations when the same input is provided.
- Get the worker process ID: This retrieves the process ID of the current worker process and prints it for reference.
- Measure the start time for processing: This records the current time before starting the inference process.
- Set up parameters and constants for face and mouth detection: These variables include font type, text size, font colors, thickness of the text, font scale, and threshold for binary conversion.
- Obtain absolute file paths for the XML cascade classifiers: These lines construct absolute file paths for the XML files used by the Haar cascade classifiers for face and mouth detection.
- Convert the input image (provided as a byte stream) to a NumPy array: The image byte stream is converted into a PIL Image and then into a NumPy array. The image is also converted to grayscale for face detection.
- Apply thresholding to create a binary image: This step helps in detecting faces more effectively.
- Detect faces in both the grayscale and binary images: Cascade classifiers are used to detect faces in both the grayscale and binary images.
- Handle cases where no face is found: If no faces are detected in both images, a message is added to the image, and the condition is set accordingly.
- Handle cases where only one face is found in the binary image: If only one face is found in the binary image, it assumes a masked face.

- Process cases where faces are found in the grayscale image: For each detected face, a rectangle is drawn around it, and a region of interest (ROI) is extracted in grayscale.
- Detect mouth rectangles within the face region: Using a mouth cascade classifier, it attempts to detect mouths within the face region.
- Check if no mouth is detected: If no mouth is detected, it assumes a masked face.
- Process cases where mouths are detected: If mouths are detected within the face region, it assumes a face without a mask and updates the condition accordingly.
- Measure the end time for processing and calculate the processing time: This records the current time after completing the inference process and calculates the total processing time.
- Print the completion message and return the condition and processing time: The completion message, including the worker process ID, is printed, and the condition along with the processing time is returned as the result of the inference process.

II. Creating inference for API (**inference.py**):
- Import required modules and functions: Here, necessary modules from FastAPI, PIL (Python Imaging Library), urllib, and a function named process from the mask_inference module are imported.
- Create an APIRouter instance: This initializes an instance of the FastAPI APIRouter, which allows you to define routes and handlers for your API.
- Define a recursive function to count values in a nested dictionary or list: This function takes an object (obj) as input and recursively counts the values in nested dictionaries or lists.
- Define a route for handling POST requests to "/inference":
  This route handles POST requests to "/inference" and accepts three parameters:

  - file: An optional UploadFile parameter for uploading an image file.
  - image_url: An optional string parameter for providing a URL to an image.
  - model_in_use: A string parameter with a default value of 'mask' specifying the model to use.

- Initialize variables and check input conditions: Here, it checks whether file or image_url is provided. If file is provided, it checks whether the file type is JPG; if not, it returns an error. If image_url is provided, it fetches the image and checks its file type. If none of these inputs are provided, it sets result to indicate that no input was provided.
- Process the image based on the chosen model: If a file is provided, it reads the image, checks the chosen model, and calls the process function with the image data. The processing_time variable is also recorded.
- Print processing time and return the result: The processing time is printed to the console, and the result is returned as a response.

b. Build endpoint for API:
   - Import necessary modules and functions: This code imports the FastAPI class and CORSMiddleware from FastAPI for creating the API and handling Cross-Origin Resource Sharing (CORS). It also imports the inference module, presumably containing routes related to machine learning inference.
   - Create an instance of the FastAPI class: This line creates an instance of the FastAPI class named app. It provides custom paths for the OpenAPI JSON file (openapi_url) and the API documentation (docs_url).
   - Add CORS middleware to the FastAPI app: The CORSMiddleware is added to the FastAPI app, allowing cross-origin requests. In this case, it is configured to allow any origin ("*"), any HTTP method, any headers, and credentials.
   - Include the router from the inference module: The router from the inference module is included in the app with a specified prefix (/api-inference/v1/ml) and tags. This means that the routes defined in the inference module will be accessible under this prefix.
   - Define a root endpoint: This is a simple endpoint that responds to HTTP GET requests to the root path ("/"). It returns a JSON response with a message indicating that it's an ML API.

c. Run code in folder API:
   I. Access command line, make sure the path is in the main folder, and then change to api folder with this command: **cd api**.
   II. Run endpoint code with this command: **uvicorn endpoints:app --workers 4.**
   III. Access API, input image, and get the prediction use this adddress: **http://127.0.0.1:8000/api/v1/ml/docs.**

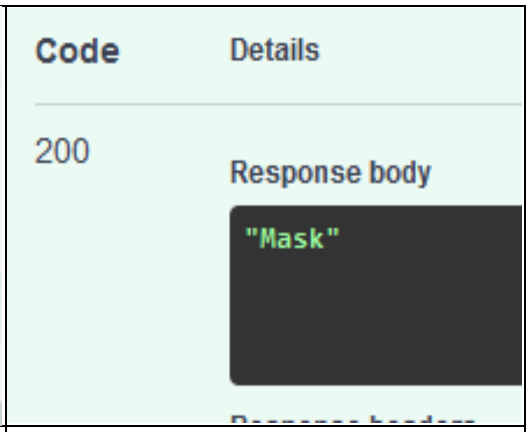
d. Run API code with Docker (CPU Version):
   I. Create a docker image with the following steps:
      - Base Image Selection**:** This line specifies the base Docker image to use. In this case, it's based on Python 3.7 with a slim Debian-based image (**python:3.7-slim**).
      - Working Directory: Sets the working directory inside the container to /code. This is where the subsequent commands will be executed.
      - Copy Requirements File**:** Copies the requirements.txt file from the host to the working directory inside the container.
      - Update and Install Dependencies**:** Updates the package index and installs additional system dependencies (ffmpeg, libsm6, libxext6) using apt-get.
      - Install Python Dependencies**:** Uses pip to install Python dependencies listed in requirements.txt. The --no-cache-dir option avoids caching the downloaded packages.
      - Create User**:** Creates a user named "user" with a user ID of 1000. The -m flag ensures that a home directory is created for the user.
      - Switch to User Context**:** Switches to the "user" user, ensuring that subsequent commands are executed as this user.
      - Set Environment Variables**:** Sets environment variables for the user's home directory and adds the user's local bin directory to the PATH.

- Set Working Directory**:** Changes the working directory to /home/user/app inside the container. This is where the application code will be placed.
- Copy Application Code: Copies the application code from the host to the working directory inside the container. The --chown=user option ensures that the files are owned by the "user" user.
- Command to Run the Application: Specifies the default command to run when the container starts. In this case, it uses uvicorn to run the FastAPI application (endpoints:app) on host 0.0.0.0 and port 7860 with 4 worker processes.

II. Build Docker Image
- Before building the image with the Docker command, make sure the path is in the api folder and then, follow this command to build image: **docker build --tag izzansilmi/sfmd-ml:1.0.5 .**

III. Run Docker Container
- Follow this command to run docker container: **docker run -it --name sfmd-ml -p 7860:7860 izzansilmi/sfmd-ml:1.0.5.**

e. Deploy to Remote Server (Hugging Face):
I. Create New Space in **https://huggingface.co/spaces**. Select deploy using Docker.
II. Commit and Move all the code in the API folder to space. Docker will deploy automatically.
III. API will be accessible via URL, Example: **https://isa96-fm-ml.hf.space/api/v1/ml/docs.**

	
Input image	Output From API

5. About Image

The base image used to create the dockerfile is python: 3.7-slim. If you use a lower version of the image then it may be difficult to support requirements. When using the image version above, some can do it and some can't. So it can be concluded that if the

image can install the dependencies in the requirements, it means the code will be safe to deploy.

6. Test Solution

In the previous explanation, four methods were used to test the model. Firstly, the inference.py code can be employed, requiring the input image path, and the prediction results can be observed in the output folder. Secondly, the API can be utilized by executing the endpoint.py code, inputting the image, and receiving the result as feedback in the form of predicted string data. The third method is similar to the second; the only difference is that when you run the Dockerfile, the output is also in the form of string data. Lastly, in the fourth method, both the Dockerfile and the code in the API folder are placed on the remote server, and the output is likewise presented as string data.

7. Suggestion

As for optimizing the solution, here are some suggestions:

a. Add appropriate datasets: Make sure the dataset you use includes a variety of people with and without masks. A good and representative dataset will help the model to learn better.
b. Image Preprocessing: Consider applying image preprocessing techniques such as resizing, normalization, or histogram equalization to improve the performance of the face detection algorithm.
c. Addition of Data Augmentation: Apply data augmentation to the dataset to create more variety. This can include rotation, flipping, zooming, and brightness changes. Data augmentation helps models to better handle variations that may occur in the real world.
d. Hardware Acceleration: Depending on your platform, consider using hardware acceleration (e.g., OpenCV with CUDA support) to speed up face detection.
e. Parallelization: If you are processing multiple images, explore parallelizing the face detection process to improve overall performance.
f. Better Face Detection Model: Replace your face detection model with a better or newest model. As time goes by, there may be new models that have better performance.