40102820

‘Building a Computer Vision System using Open AI CLIP’

A dissertation submitted in partial fulfilment of
The requirement for the degree of
MASTER OF SCIENCE in Software Development
in
The Queen’s University of Belfast
By
‘Noel Proctor’
‘03/05/2023’

Declaration of Academic Integrity

Before signing the declaration below please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook.
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic).
3. Does not exceed the specified page limit.
4. Is clearly presented and proof-read.
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.
6. Software and files are submitted via Canvas.

I declare that I have read both the University and the School of Electronics, Electrical Engineering and Computer Science guidelines on plagiarism http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiarism/ - and that the attached submission is my own original work.

No part of it has been submitted for any other assignment and I have acknowledged in my notes and bibliography all written and electronic sources used. I am aware of the disciplinary consequences of failing to abide and follow the School and Queen’s University Regulations on Plagiarism.

Name: Noel Proctor
Student Number: 40102820

Acknowledgements

I would like to thank Dr John Bustard for providing regular contact and much valuable advice throughout the project.

Abstract

Computer vision is a field of artificial intelligence that involves deriving meaningful information from digital images, videos, and other visual inputs. One of its key applications is image classification, which involves training a machine learning model to identify and categorize images into predefined classes. However, traditional models come with limitations, such as being task-specific and requiring a separate model for each new classification task. OpenAI's CLIP model has addressed these limitations by being flexible and capable of recognizing a broad range of visual concepts. This project aims to create a computer vision system that utilizes the CLIP model to associate metadata with human face images captured via a webcam. The system will improve its accuracy over time by comparing the CLIP-attached labels to a database of images with verified labels. A front-end application and back-end APIs using Flask and Python will be developed to achieve this functionality. The APIs include Face Detection, Image Embedding Generation, and Labelling APIs.

Table of Contents

Chapter 1: Understanding The Problem

Background

Computer vision is “a field of artificial intelligence that enables computers and systems to derive meaningful information from digital images, videos, and other visual inputs…” (IBM, *‘What Is Computer Vision’*). One of its key applications is image classification, where a machine learning model is trained to identify and categorize images into predefined classes. For instance, it can be used to label an X-ray image as cancerous or not, or to categorize different breeds of dogs. Image classification has also been used in the self-driving car industry where image classification models have been trained to recognise objects such as cars, pedestrians, lampposts, and road signs (Boesch, 2023). However, traditional image classification models come with limitations. They tend to be task-specific in that they are good at performing one task and one task only. This means that a separate model must be trained for each new classification task, which can be both time-consuming, expensive (computationally and financially) and require huge amounts of data (Howard & Gugger, 2020).

In January 2021, Open AI released a deep learning model called CLIP *(Contrastive Language-Image Pre-Training).* The model was trained on the text paired with images found across the internet and is therefore capable of recognising a broad range of visual concepts (Radford et al., 2021). This has resulted in a model that is flexible and does not need to be retrained for each classification task. The CLIP model performs exceptionally well on zero-shot classification tasks, matching the performance of models such as ResNet-50 and ImageNet, without needing to be trained on the same large datasets that the aforementioned models were trained on**.** (Radford et al., 2021). It should therefore be possible to develop a computer vision system using CLIP that can label a wide range of visual concepts found in images without needing to be retrained.

This project aims to develop a system that utilizes CLIP to associate text labels with images of human faces. By leveraging large, labelled datasets such as the Celeb-A dataset (Liu, Luo, Wang, and Tang 2015) and the Fairface dataset (Karkkainen, Kimmo, and Joo, Jungseock, 2021), we can test and experiment with the system at scale, identifying any potential problems and areas for improvement. Focusing on human faces provides a unique label set for experimentation, allowing us to measure the system's accuracy in associating age, gender, and race labels with images of human faces. We can also evaluate the system's accuracy in predicting each label within a set. Once a base set of labels has been established and results from experimentation are recorded, the work can be expanded to include less objective label sets like emotions or perceptions or even extend it to other image categories.

Project Goal

This project seeks to create a computer vision system that uses CLIP to associate labels to images of human faces, and which can become more accurate over time. This will be done in three distinct phases.

Phase one aims to develop a UI tool which will simulate images being collected and labelled by a computer system. The tool should be able to access a user's webcam and detect if a human face is present in the image. Any faces detected should be cropped and labelled by the system. Phase one will have no method of label correction to improve accuracy. The accuracy of label predictions will then be tested at scale.

Phase 2 seeks to develop at least 1 method of improving the accuracy of predictions over time. To do this the UI tool should allow users to verify predicted labels as accurate or inaccurate. The results should then be saved to an image database, which the system will use to improve the accuracy of label predictions. As the image database grows it will contain a wider range of visual concepts. As a result, the closest image embedding in the database to the image being labelled will become increasingly closer over time, with the expectation that prediction accuracy will therefore improve. In the case of a dataset of human face images, we can use CLIP to match age and race labels and measure the system's initial accuracy. Then as more images of individuals from diverse age and race groups are added to the database, the system will be able to compare the current image with similar images from that specific group stored in the database.

Assuming the project has progressed smoothly up to this point and if there is enough time, phase 3 will seek to expand the system's capabilities in two ways. First, by allowing users to add their own labels to images. For example, if the system has labelled an image as "White, 20-30, Male", the user should be able to add the label "wearing glasses, wearing a hat" if the person in the image is indeed wearing a hat or glasses. Secondly, phase 3 will aim to expand the range of object categories that the system can detect and label beyond just human faces. For example, images of dogs or plants. Expanding the system's object recognition capabilities will enable users to apply the technology to a wider range of use cases.

To achieve this, a front-end UI Tool and several back-end services will be developed. The UI tool will need to be able to access a user's webcam via a web browser and send it to the back-end API for processing. HTML and JavaScript will be used to build this functionality, being the most common languages used to develop web-based applications that can access a user's webcam. There is the option to use a framework such as React; however, it is not necessary for the functionality required in this project as it will rely mainly on server-side functionality. The back-end services will be developed using Python and the Flask framework. Python is a widely used programming language in data-driven and machine-learning projects, having a vast number of libraries that can support development of all phases. Flask is a lightweight framework that comes with a built-in development server and a more manageable learning curve than full-stack frameworks like Django. (Klein, 2021).

Three Flask-based APIs will be developed to provide the necessary functionality. This approach provides a higher degree of modularity, makes troubleshooting easier, localises errors, and provides a dedicated server for each function allowing for performance optimisations.

The APIs needed are:

1. Face Detection API.
2. Image Embedding API.
3. Labelling API.

Face Detection API

The Face Detection API will be responsible for detecting the presence of faces in an image. It will need to receive an image and return the coordinates of any face detected. DeepFace and Retinaface are two lightweight face analysis frameworks that can potentially achieve this (Serengil et al., 2020) While DeepFace has a prebuilt API, it does not provide the needed functionality. However, the DeepFace.FaceDetectors class offers the ability to both detect a face and return its coordinates (Serengil, 2023). Retinaface provides similar functionality and is regarded as being more accurate, however, it is slower than DeepFace (Deng et al., 2019).

Image Embedding API

The image embedding API will need to receive an image and generate an image embedding, which can later be used for comparison against text embeddings. As per the CLIP documentation, a compatible version of PyTorch will need to be installed in the environment where this API is running (See documentation @ https://github.com/openai/CLIP). PyTorch requires a Python version of 3.9 or below, therefore this will be the Python version used throughout the project to ensure compatibility.

Labelling API

The Labelling API will be responsible for matching the closest text labels to the image embedding it receives. The CLIP module alone can be used for this process, but when dealing with a large number of labels it may be more efficient to use the FAISS library for nearest neighbour search. FAISS is an open-source library developed by Facebook for “efficient similarity search and clustering of dense vectors” and can be used to build an index of all embeddings in the verified image dataset (FAISS Documentation, 2023). By indexing the embeddings in the database and using the index.search() function, we can efficiently search through a large database of image embeddings to retrieve the index of the most similar embedding to the image being labelled (Patil, 2019). Both methods will require experimentation to determine the most effective solution. A final requirement of the

Labelling API is that it should also be able to connect to and search a database of verified images, and use the labels attached to the closest matching image to improve the accuracy of results.

Requirements

Requirement 1: Webcam Access

Priority: Must Have

- The system should be able to access the users' webcam through a web browser, enabling them to view a live video stream on the screen.
- There should be a button for each labelling method developed below the video stream. There should also be a "Stop" button.

Acceptance Criteria:

1. When a user visits the designated web address (localhost:5000/analyser), they will be prompted to grant permission for the system to access their webcam.
2. If the user grants permission, the system will display a live video feed of the webcam stream on the screen. There will be a button for each labelling method and a "Stop" button which stops any labelling process which is in progress.
3. The page should be available to all users.

Requirement 2: Button Functionality

Priority: Must Have.

• Each labelling method button should begin the labelling process for that method when clicked. All other labelling method buttons should be disabled until the "Stop" button is clicked.

• The "Stop" button should stop any labelling process currently in progress.

Acceptance Criteria:

1. Upon clicking on any of the labelling buttons, the system should initiate the labelling process for that labelling method.
2. Each labelling button should be named "Method 1", "Method 2" etc.
3. The user should only need to press a labelling method button once, which should start the labelling process. In case of any error, the system should automatically attempt to restart the process.
4. When the labelling process is in progress, all labelling method buttons should be disabled.
5. The "Stop" button should halt any process that is in progress and re-enable the labelling method buttons.

Requirement 3: Face Detection (Single Face)

Priority: Must Have

- The system should be able to detect when a face is present in an image.
- When a face is detected, the system should crop the face and save the cropped image.

Acceptance Criteria:

1. When a face is detected in an image, the system should crop the face and save the image for further processing (i.e., generating an image embedding).
2. In case a face is not detected in the image, the system should continue to sample frames from the webcam and repeat the face detection process until a face is detected.

Requirement 4: Face Detection (Multiple Faces)

Priority: Should Have

- The system should be able to detect the presence of multiple faces in an image.
- Each face detected should be cropped from the image and saved for processing (generating a clip image embedding).

Acceptance Criteria:

1. For each face detected in the image, the face is cropped and saved for processing (generating an image embedding).
2. If no face is detected in the image, the system should continue to sample frames from the user's webcam and repeat the process until a face is detected.

Requirement 5: The system should draw a border around each face detected in the image.

Priority: Must Have

- The system should draw a border around each face detected in the image.
- Each face detected should be labelled for future identification. For example, if 1 face is detected it should be labelled "Face 1". If 2 faces are detected, they should be labelled "Face 1" and "Face 2" etc. This is to allow the user to identify which faces the labels have been applied to.

Acceptance Criteria:

1. For each face detected, a red border should be drawn around the face in the original image.
2. Each face should be labelled.
3. Labels applied to faces should not be cut off by image borders or other faces.

Requirement 6: Generate Image Embeddings

Priority: Must Have

- The system should be able to generate a clip image embedding for each face detected.

Acceptance Criteria:

1. The system can generate a clip embedding for each face detected in an image captured by a user's webcam.

Requirement 7: Compare Image embeddings and text embeddings.

Priority: Must Have

- The system should be able to compare the clip image embedding generated in requirement 6, against a set of clip text embeddings of labels that should be applied to the image. The closest matching label from each set of text embeddings should be saved so that they can later be associated with the image.

Acceptance Criteria:

1. When an image embedding is generated, the system can match the closest text embedding from a set of labels to that image.

Requirement 8: Associate labels to the image and display them to the user.

Priority: Must Have

- The system should display to the user the cropped facial image beside labels that clip has attached to that image. The system should also display to the user the original image with borders drawn around each face detected (See Requirement 5)

Acceptance Criteria:

1. When a labelled image has been received, it should be displayed to the user.
2. The original image should also be displayed to the user, with borders drawn around each detected face.

Requirement 9: Label Self-Correction

Priority: Should Have

- There should be at least 1 method by which the system can correct incorrect predictions.
- The accuracy of the system should increase over time as more images are added to the database. Prediction accuracy for any label correction method developed should be measured and recorded in "Chapter 4: Experimentation".

Acceptance Criteria:

1. There should be at least 1 method of correcting incorrect label predictions.
2. The accuracy of any label correction methods developed should be measured and recorded in Chapter 4.

Requirement 10: Automatic restart.

Priority: Must Have

- Once a face has been detected and a labelled image displayed for the user, the system should automatically begin the process again.

Acceptance Criteria:

1. When a labelled image is returned to the user, the process automatically begins again.
2. The process should only stop when the user exits the webpage or presses the “Stop” button (see Requirement 2).

Requirement 11: Label Verification

Priority: Should Have

- There should be a screen that allows registered users to verify labels attached to images.
- The screen should only be accessible to registered users.
- The screen should show the next image from the database that has not been verified. It should display the labels that have been associated with the image and provide a method for the user to confirm or reject these labels.

Acceptance Criteria:

1. The screen should only be accessible to registered users.
2. The screen should retrieve the next unverified image from the database. On the left of the screen the image should be displayed. On the right of the screen, all unverified labels should be displayed.
3. There should be a check box for each label.
4. There should be a “Confirm Labels” button below the labels. When the user clicks this button, the user’s selection should be saved, and the user should be shown a screen that allows them to add their own labels to the image (see requirement 12 below)

Requirement 12: User-Added Labels

Priority: Nice To Have

- Registered users should be able to enhance the image labelling process by adding more labels to the existing set of labels generated by the system.
- This should occur after label verification (see requirement 11).

Acceptance Criteria:

1. After verifying the labels generated by the system, a registered user should be able to add additional labels to the image.
2. Users can add 0 or more labels to an image as they see fit.

3. Users should be able to remove any additional labels they have attached before submitting the image. (I.e., in case of spelling mistakes or if they have reconsidered the label).
4. The user should not be allowed to add empty labels.
5. There should be a submit button that allows the user to submit additional labels to the database. The record for the image should be updated with the field "UserAddedLabels" that contains the labels added by the user.

Requirement 13: User Login

Priority: Must Have

- There should be a user login page that allows registered users to log into the system.
- On the page there should be a form with the fields "Username" and "Password".
- There should be a "Login" button that logs the user into the system if their details are recognised.

Acceptance Criteria:

1. When the user enters a recognised username and password and clicks submit, they should be logged into the system as a registered user.
2. When the user enters incorrect details and clicks submit, an error message saying "Invalid Credentials" should be displayed.

Requirement 14: User Roles

Priority: Must Have

- The system should have 3 user roles. Anonymous user. Registered user. And Admin user.

Acceptance Criteria:

1. Admin users should have access to all pages on the application and have access to all functionalities.
2. Registered users should have access to the "Analyse", "Verify" and "Login" pages.
3. Anonymous users should have access to the "Analyse" and "Login" pages and functionally

Requirement 15: User Creation

Priority: Must Have

- There should be a user creation page that is accessible only to admin users.
- There should be a form on the screen that allows the admin to create a new user. The form should have the fields "Username", "Password", and an "isAdmin" checkbox.
- There should be a submit button at the bottom of the form which should create a new user if the details entered are valid. If a valid user is created, the message "User Successfully Created" should be displayed on the screen.

- If a user already exists, the message “User already exists with this username” should be displayed on the screen.

Acceptance Criteria:

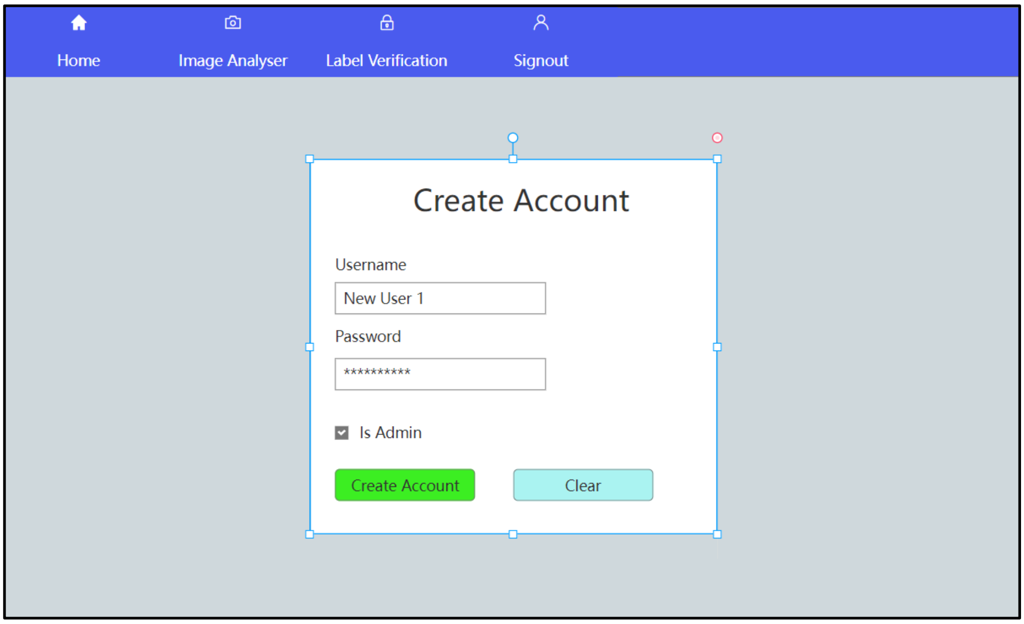
1. The “Account Creation” page should be accessible to admin users only.
2. There should be a form containing the fields “Username” and “Password”. There should also be a checkbox labelled “Is Admin”.
3. Username and password cannot be blank or greater than 50 characters.
4. When the user clicks the “Submit” button, the new user details should be validated. If the credentials are valid, a new user should be created.
5. If the details are invalid, an appropriate error message should be displayed (i.e. “Password cannot be blank”).
6. If the new user details are valid but a user with the same username already exists on the system, the message “User already exists with this username” should be displayed.

Chapter 2: User interface design

The UI of this project has a minimalist and functional design. Its function is as a UI tool to support collecting images for label identification and verification. The application consists of five screens, each with specific user access permissions. Mock Plus was used to create all mock-ups below. The website is available at https://www.mockplus.com/.

Account Creation Page

Available only to admin users, this screen allows for the creation of new user accounts.



Create Account

Username: New User 1

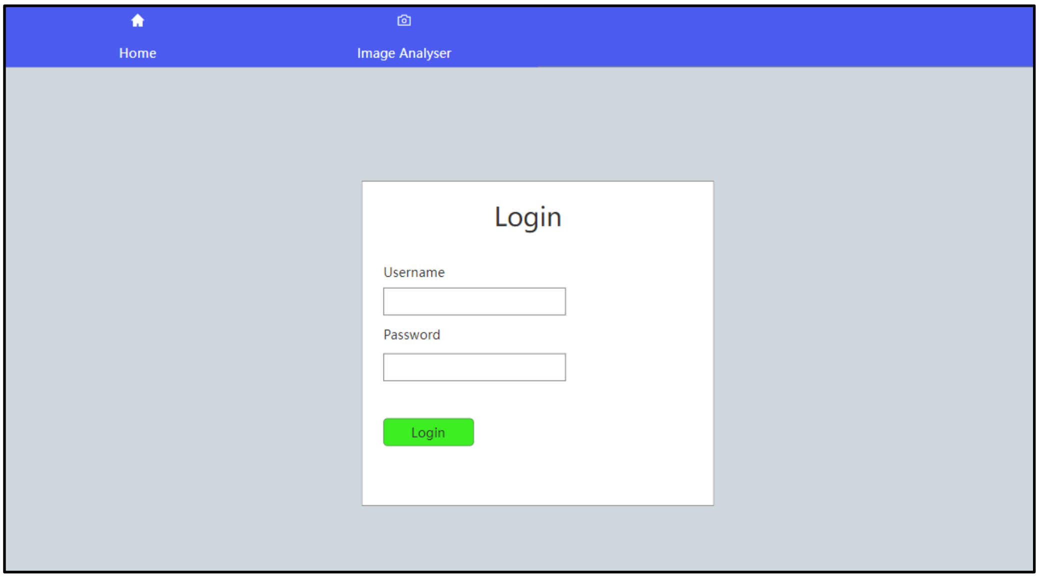
Password: **********

☑ Is Admin

Figure 1: Account creation screen

Login Page

Available to all users. This screen allows users to log into their accounts.



Login

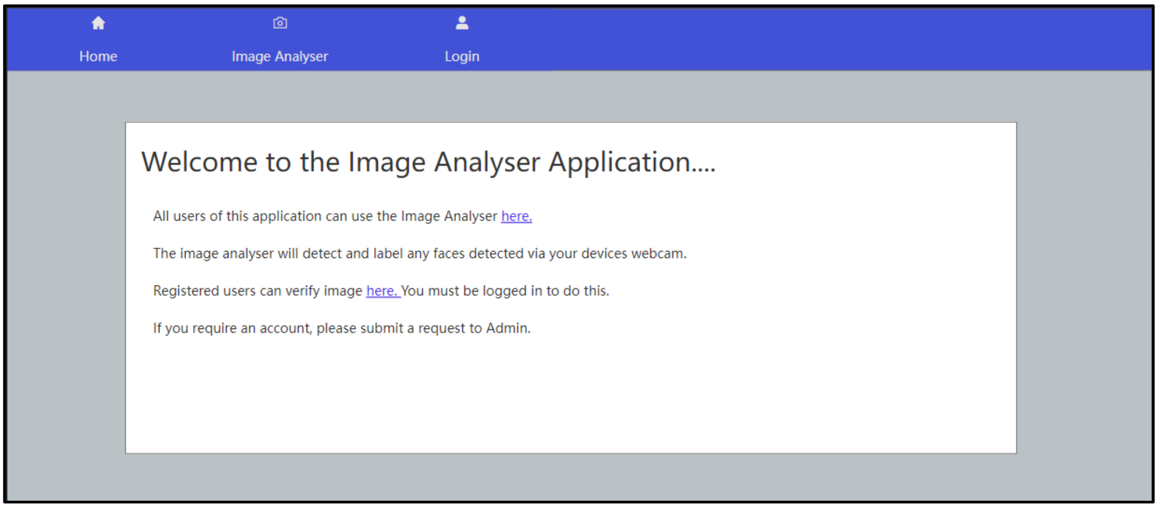
Username

Password

Figure 2: User login screen

Home Page

Available to all users. This screen displays a description of the system and its functionalities.



Welcome to the Image Analyser Application....

All users of this application can use the Image Analyser here.

The image analyser will detect and label any faces detected via your devices webcam.

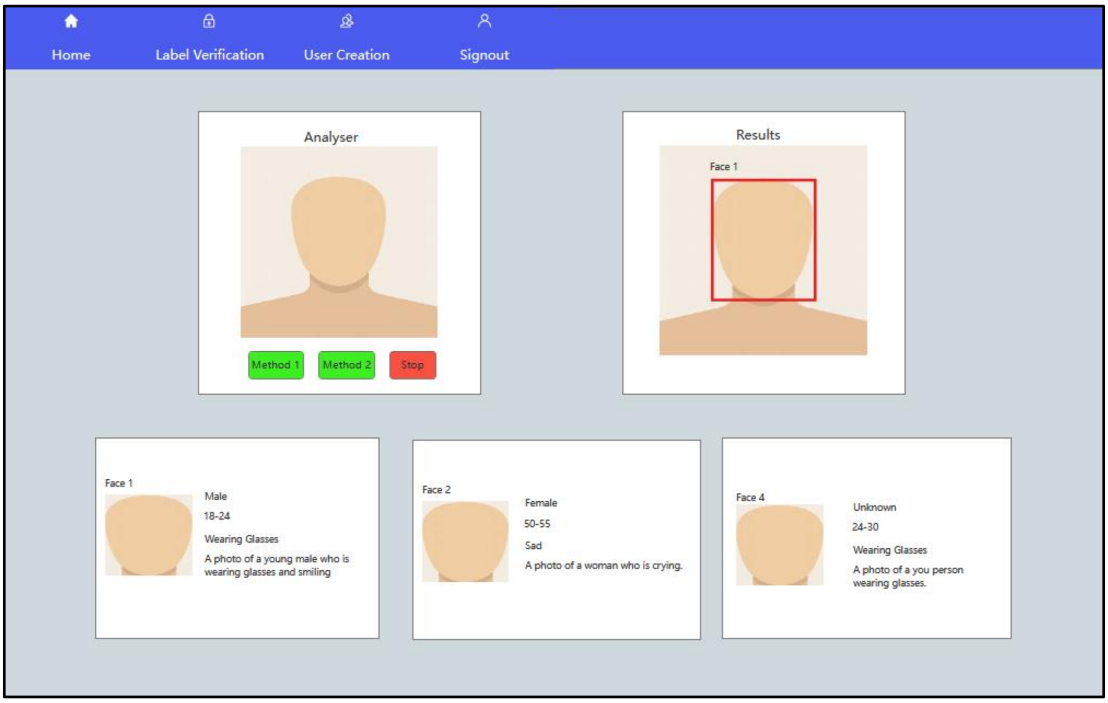
Registered users can verify image here. You must be logged in to do this.

If you require an account, please submit a request to Admin.

Figure 3: Home screen

Image Analyser Page

Available to all users. This screen detects faces through a user's webcam and applies labels to each face detected. The left-hand card labelled "Analyser" shows the user a live stream of their webcam feed. The right-hand card has a box which shows the image with any detected faces outlined with a red border. The labels that have been associated with each face detected are displayed in the boxes below along with a cropped image of the detected face.



Face 1
Male
18-24
Wearing Glasses
A photo of a young male who is wearing glasses and smiling

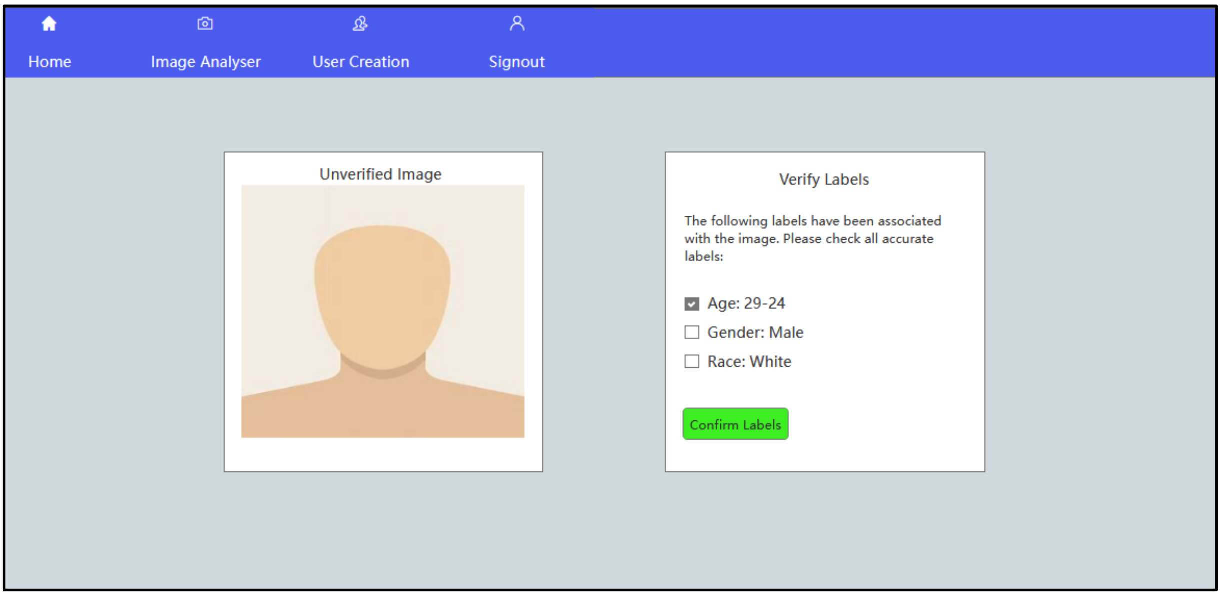
Face 2
Female
50-55
Sad
A photo of a woman who is crying.

Face 4
Unknown
24-30
Wearing Glasses
A photo of a you person wearing glasses.

Figure 4: Image analyser screen

Label Verification Page

Available to registered users only, this screen allows a registered user to verify image labels. If an unverified image exists in the database, it is retrieved and displayed on the left-hand side of the screen. The labels that have been associated are displayed on the right-hand side of the screen. The user checks the box of each accurate/ correct label.

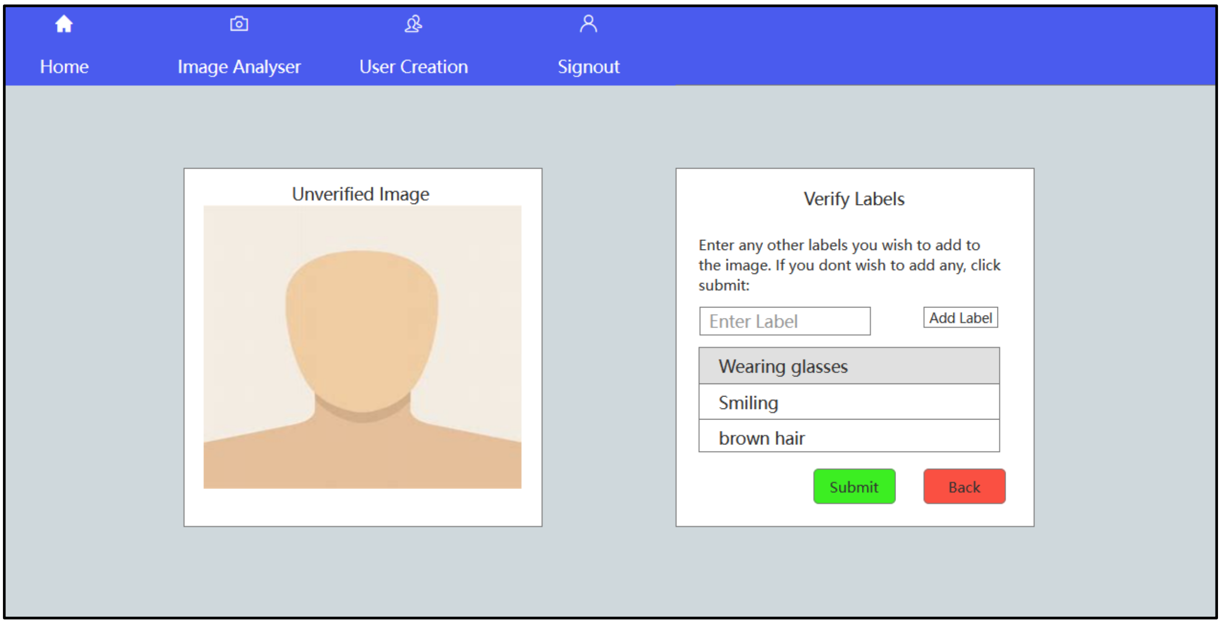


The following labels have been associated with the image. Please check all accurate labels:

☑ Age: 29-24
☐ Gender: Male
☐ Race: White

Figure 5: Label verification

When the user clicks “Confirm Labels”, user selection is saved and the user is shown a form which allows them to add their own labels, as shown below in Figure 6. When the user clicks the submit button, labels that have been checked are saved as “verified_labels”. Labels that are unchecked are saved as “incorrect_labels”. User-added labels are saved as “UserAddedLabels.



Enter any other labels you wish to add to the image. If you dont wish to add any, click submit:

Figure 6: Label verification, user-added labels

Chapter 3: Architecture Design and algorithm explanation

System Architecture

The system architecture is shown in *Figure 7* below. The front-end UI tool is a Flask-based web application which communicates with three Flask-based APIs - the Face Detection API, Image Embedding API, and Labelling API. The backend server of the UI tool is connected to an image database, where newly analysed images are saved and updated after label verification. The labelling service also connects to the same image database to improve the accuracy of label predictions by comparing predicted labels to the verified labels of existing images. These APIs work together to create an image labelling system that can learn and improve the accuracy of results over time (see Chapter 4 for experimentation results).

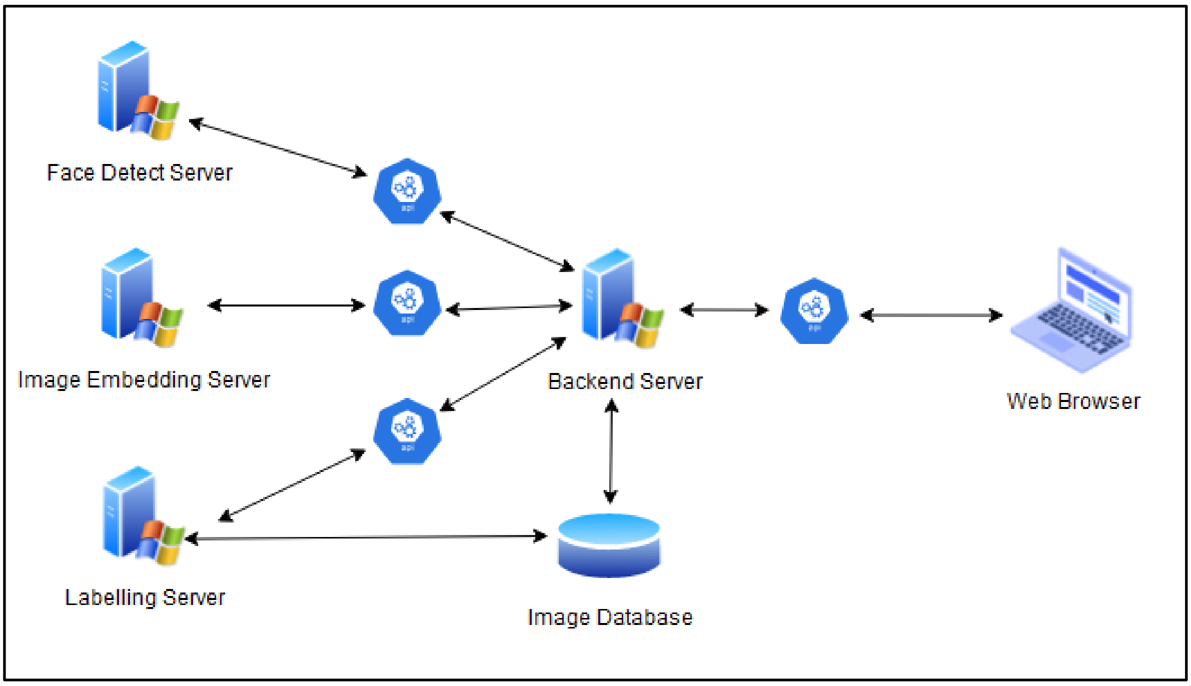


Figure 7: System architecture diagram

Image Analyses Process Flow

Image analyses begins when the user clicks either of the “Method 1”, “Method 2”, or “Method 3” buttons on the Image Analyser screen. Each button triggers a process that captures an image and employs a different labelling method to predict labels.

First, the system captures a screenshot of the current frame and sends it to the backend server for processing. The Backend server validates the image data and, if the image data is valid, a request is made to the Face Detection API to detect any faces in the image. If no faces are detected, a response is sent back to the front-end application indicating that no faces were detected, and the process restarts automatically.

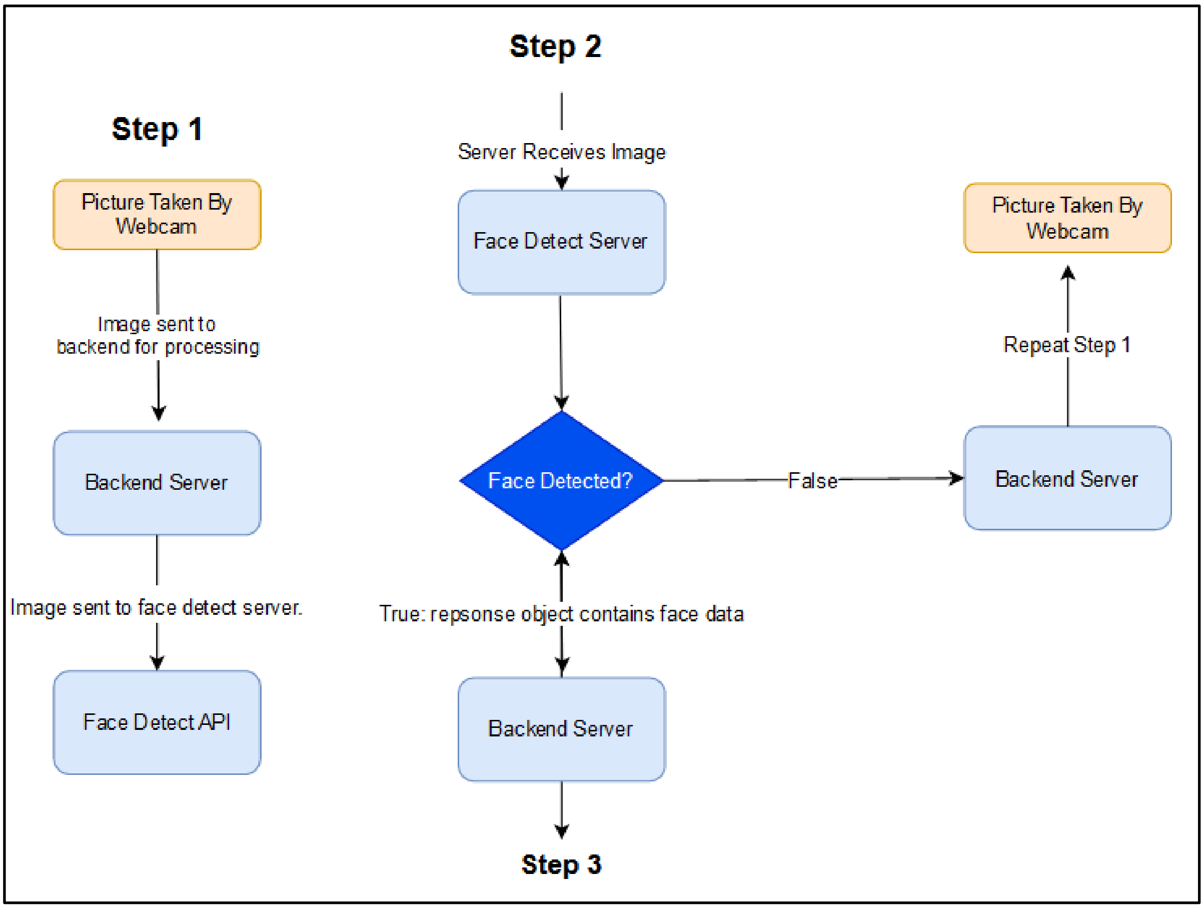


Figure 8: Image analysis flow steps 1 & 2

When one or more faces are detected, the face detection server returns the coordinates of each face in the image, along with a cropped image of each face, to the backend server of the UI tool. The backend server then saves this data to a dictionary object and sends the image data for each face to the image embedding server. The image embedding server receives the image data, validates it, and generates and responds with a clip embedding for each image. The backend server then receives these embeddings and saves them to the dictionary object created for each face in the previous step.

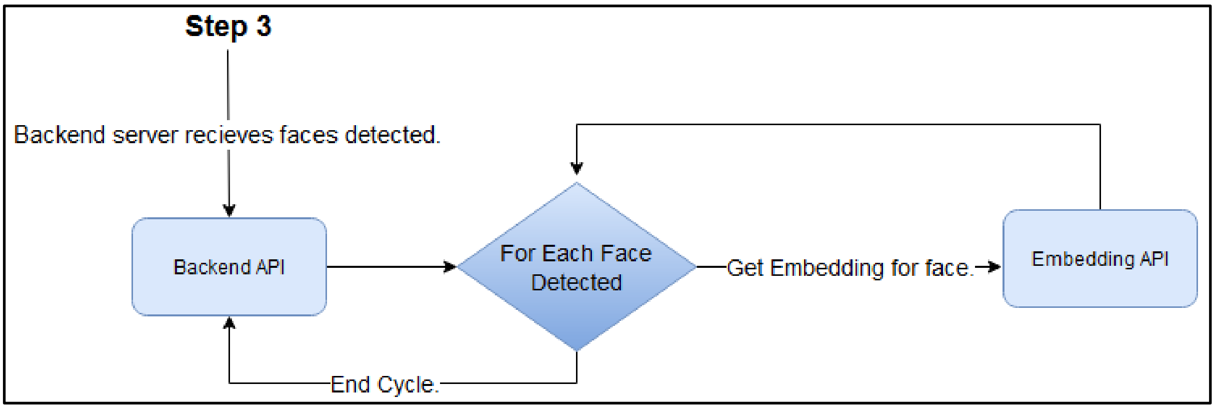


Figure 9: Image analysis Step 3

The embedding for each face is then sent to a specific endpoint of the labelling server based on which button the user clicked. If the user has clicked the “Method 1” button, a request is made to the “/label_method_1” endpoint of the Labelling API. If the user has clicked the “Method 2” button, a request is made to the “/label_method_2” endpoint. If the user has clicked the “Method 3” button, a request is made to the “/label_method_3” endpoint. Each endpoint uses a different approach to associate labels with an image. Each method is described in more detail in the ‘Labelling Server’ section on pages 24-27.

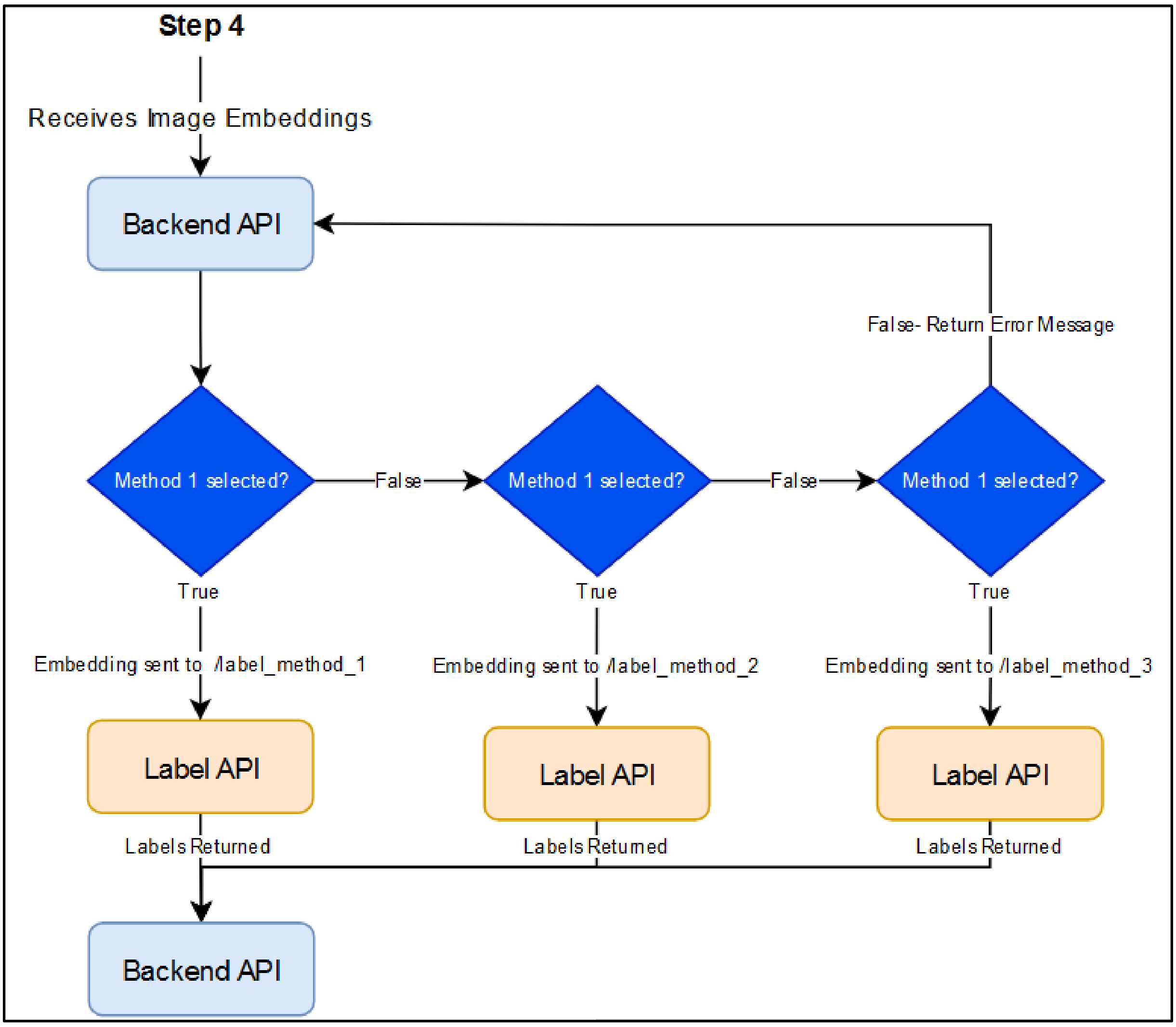


Figure 10: Image analysis flow step 4

The final step in the process occurs when the system has received the labels for each face detected in the original image. At this point the coordinates of each face are sent to the “/draw_labels” endpoint of the Labelling API. The “/draw_labels” endpoint receives the image and draws a red border around each face detected based on the facial coordinates returned by the Face Detection API earlier in the process. It then returns the modified image to the backend Server. At this point the backend server possesses all the data required and sends it to the front end to be displayed in the browser. The process then repeats until the user clicks the “Stop” button.

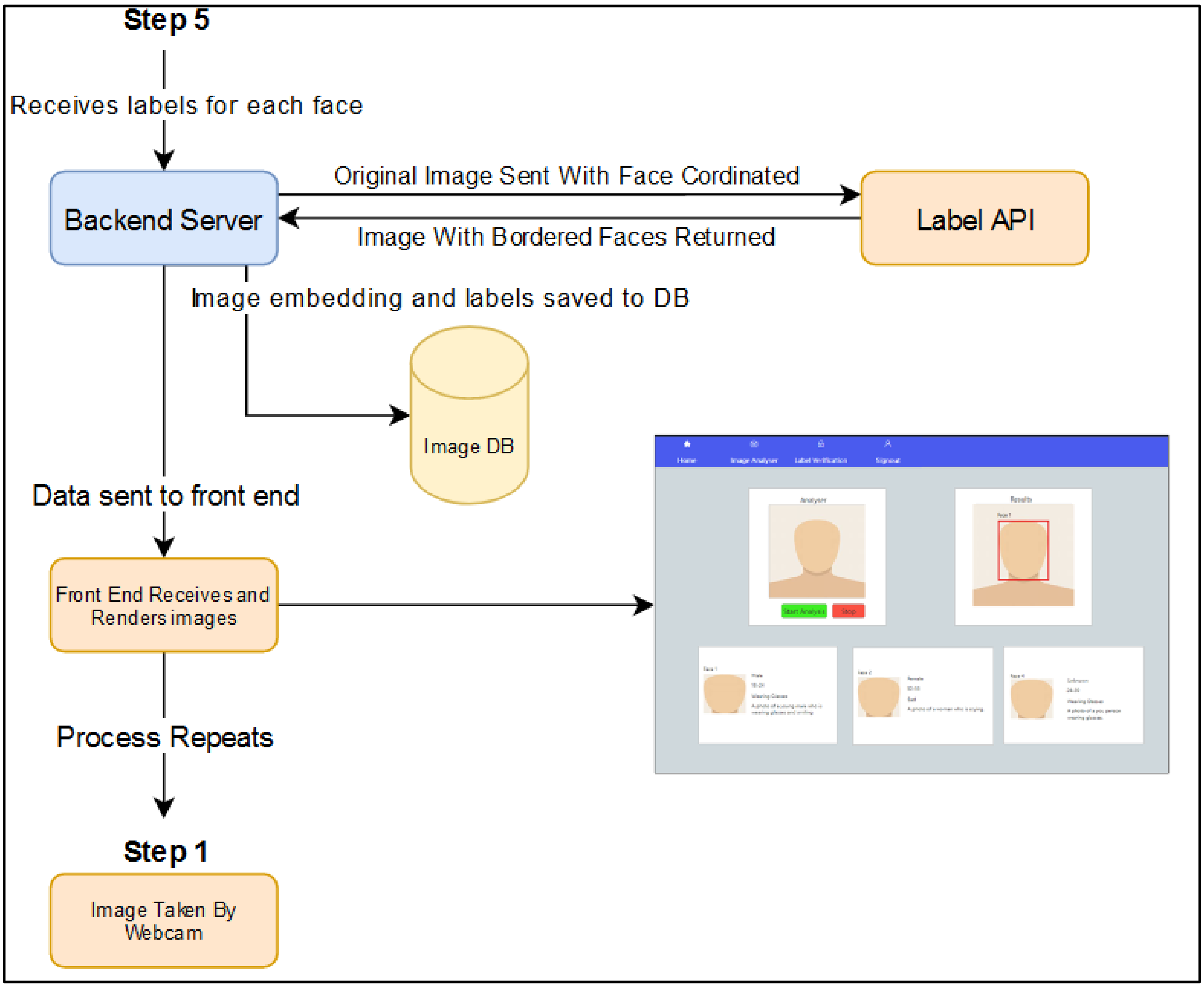

Step 5
Receives labels for each face
Backend Server
Original Image Sent With Face Cordinated
Image With Bordered Faces Returned
Label API
Image embedding and labels saved to DB
Image DB
Data sent to front end
Front End Receives and Renders images
Process Repeats
Step 1
Image Taken By Webcam
Home
Image Analyser
Label Verification
Signout
Analyser
Results
Start Analysis
Stop

Figure 11: Image analysis flow step 5

Image Verification Process Flow

The image verification process begins when a registered user logs into the application and navigates to the image verification page. If there are unverified images in the database, the system will retrieve them one by one and display them in the browser. Each image will be displayed on the left side of the browser, and the unverified labels associated with the image will be displayed on the right (as shown in the UI design on pages 11 & 12). The user then reviews the labels and selects all those that are correct before clicking the “Confirm Labels” button. The user can then add any additional labels they wish to associate with the image. When the “Submit” button is then clicked, the image database is updated with all checked labels marked as correct and all unchecked labels marked as incorrect. Any user-added labels are also added to the image record under the key “UserAddedLabels”. Finally, the "requiresVerification" flag is set to "false" to indicate that the image has been verified.

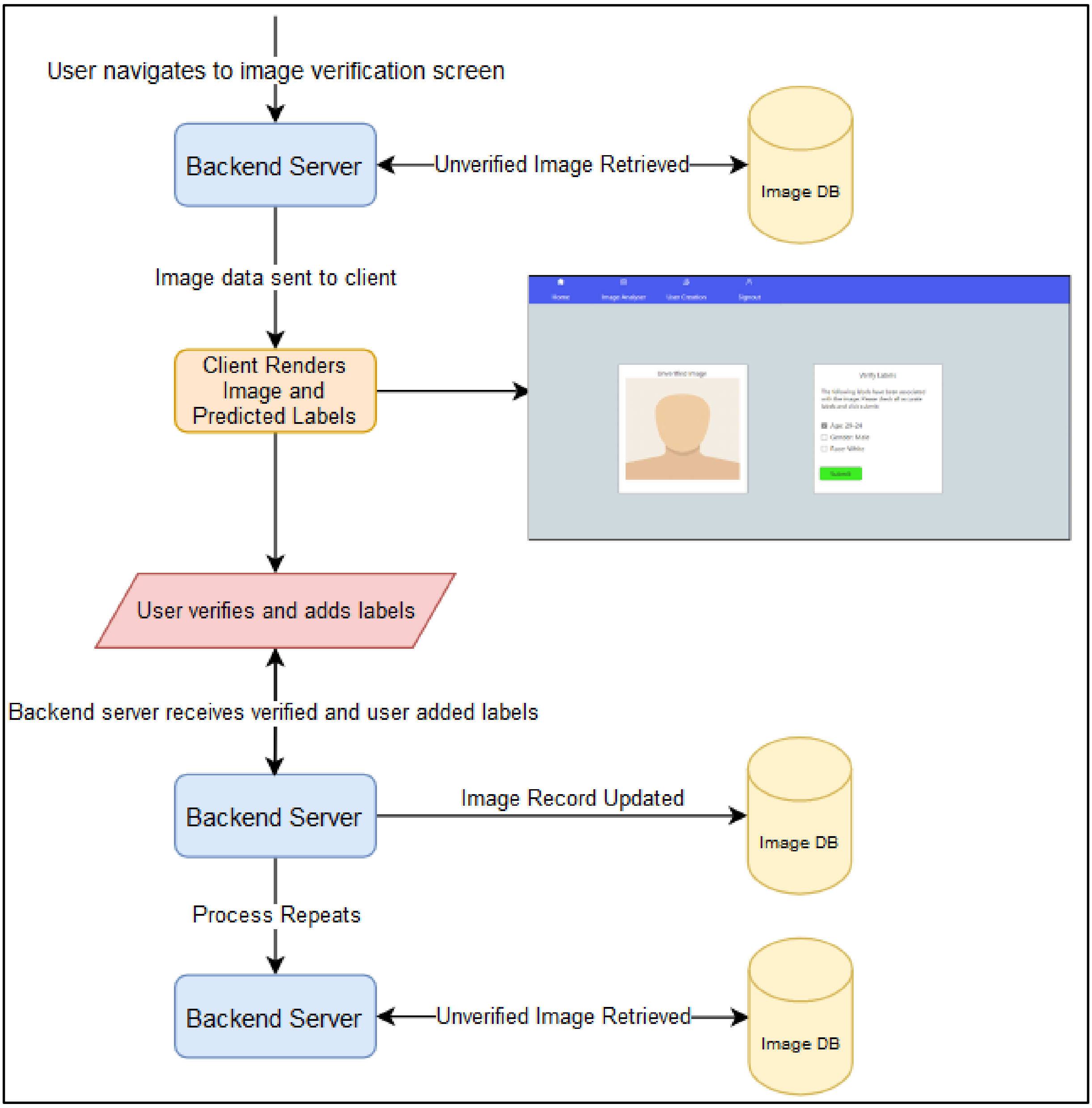

User navigates to image verification screen
Backend Server
Unverified Image Retrieved
Image DB
Image data sent to client
Home
Image Analyser
User Creation
Signout
Unverified Image
Verify Labels
☒ Age: 20-24
☐ Gender: Male
☐ Race: White
Submit
Client Renders Image and Predicted Labels
User verifies and adds labels
Backend server receives verified and user added labels
Backend Server
Image Record Updated
Image DB
Process Repeats
Backend Server
Unverified Image Retrieved
Image DB

Figure 12: Image verification process flow

UI Tool

Front End

The application takes advantage of Flask's templating capability to reuse a base HTML template across all pages. This approach allows the stylesheets, CDNs, and JavaScript files to be defined in a single centralized location (the base.html page) and applies them to all screens. The base.html page links to the Bootstrap 5 CDN (available at: https://getbootstrap.com/) to leverage its grid structures and CSS classes. It also includes a dynamic navbar that renders differently depending on the user role and their location within the application. The normalise.css stylesheet normalises UI behaviour and styles across Chrome, Edge, Firefox and Opera. I must stress that normalise.css is not my own code and is available at https://github.com/necolas/normalize.css under the MIT licence (Gallagher &

Neal, 2018). Styles.css defines the styles used throughout the application. The base.html page also links to the “static/js/jquery.js” file, which provides access to the jQuery library, and to “js/login_scripts.js”, which contains JavaScript functions used for user creation and login. These include form submit event listeners for both the create new user form and the login form. The “faceAnalyser.html” page contains the HTML code for the face analyser page and links to the “js/capture.js” file, which contains the JavaScript functions related to capturing a user image and sending it to the application back end for processing. As this is where most of the front-end logic pertaining to image labelling is found, an explanation of some of the key code from the capture.js file is found below.

The first few lines of code (Appendices: Code Snippets, Capture.js Figure 1.) in "capture.js" sets up the ability to access the user's webcam via a web browser. Two constants, “canvas” and “video” are defined to reference the respective HTML elements. This enables JavaScript to access these elements later. Constants for each of the “Method 1”, “Method 2”, and “Method 3” buttons are also defined. These are later used to enable each button when a process starts/stops.

A flag named "analyse" is defined, which is later used as a trigger to start or stop the image analysis process. The "apiUrl" constant is also defined to specify the API endpoint to which the image will be sent for processing. An empty array called "face_data" is defined, which is used to store data returned from the "/processImage" endpoint.

Navigator.mediaDevices is a Web API that provides access to media input devices such as webcams and microphones (“Getting started with media devices”, 2020). The getUserMedia() method requests access to the user’s webcam (MDN Web Docs, 2023). If the user grants access, the media stream is assigned to the source object of the video element, displaying a live stream of the user’s webcam on screen. A future improvement to this section of code would be to better handle cases where user media is not supported. For example, by displaying a message on screen rather than the console.

The stopAnalysis() function in capture.js (lines 62-66) sets the ‘analyse’ variable to false which stops the image analysis process. Take_screenshot() (Appendices: Code Snippets, Capture.js Figure 2.) sets analyse to true and begins the image analysis process. The height and width of the canvas element are set equal to the height and width of the video stream to ensure that the video frame can be drawn to the canvas without any sizing or scaling issues. Next, the drawImage() method (line 79) draws the current frame of the video onto the canvas. The resulting image is then converted to a base64-encoded string using the toDataURL() method (line 81), which allows the image data to be sent to the backend API for processing (MDN Web Docs, 2023). The fetch() method then posts the image data to the backend server endpoint defined in the apiUrl constant. If the request is successful, the response is converted to JSON using the JSON () method, and the resulting data is passed to the updateUI() function. Finally, if analyse is still set to true, take_screenshot() is called again (line 100), allowing the process to continue automatically until the user clicks on the "Stop" button, which calls the stopAnalysis() function.

The updateUI() function (Appendices: Code Snippets, Capture.js Figure 3.) is called after a successful response from the backend server. It takes the data returned from the backend server and checks if the request was successful and if a face was detected. If a face was detected, the function loops through each object in the face_data array and updates the UI accordingly. If no face was detected, the element is updated to display “No Face Detected”. To ensure that the screen does not become cluttered the function is limited to displaying a maximum of 3 faces on the screen, however, there is nothing to prevent this cap from being increased if necessary.

Backend Server

The flaskapp.__init__.py file is responsible for setting up the Flask web application. Database configuration is defined here by creating a client object connected to the "images" database hosted locally. Two decorators, login_required and admin_required, are defined here, which are used to control access to individual pages based on user roles. The run.py file imports the flaskapp.__init__.py file and defines app routes, including "/", "/analyser/", "/verify/", "/update_labels/", "/login/", "/createaccount/", and "/processimage".

The home page is rendered at the "/" app route and provides basic information about the application along with links to other pages. The "/analyser/" app route renders the faceanalyser.html page, which contains the logic described in the previous section. The "/verify/" app route renders the image verification page. The "/update_labels/" route posts the verified labels to the image database. The "/login/" and "/createaccount/" routes render the corresponding login and account creation pages.

The core image labelling logic is contained within the "/processImage/" route (Appendices: Code Snippets, ‘/processimage’ route)**.** When a POST request is made to this endpoint, the image data is extracted from the request and stored in the image_data object. A request is then made to the Face Detection API at the ‘http://127.0.0.1:5010/detect’ endpoint, which responds with the coordinates of each face detected in the image and a cropped-out image of each face (see ‘Face Detection Server’ section below). If a face is detected, a dictionary item is created for each face and stored in the face_data variable (lines 147-154), which stores the data received from each of the subsequent API requests. The “label_endpoint” variable is defined based on the labelling method chosen by the user (lines 157-165), and the face_data object is then passed into the following functions:

1. get_embeddings(): Retrieves embeddings for each face by making a request to the Image Embedding API.
2. get_labels(): Retrieves labels for each face by making a request to the Labelling API endpoint chosen by the user.

3. label_image(): Returns the original image with each face bordered by making a request to the Image Labelling API.
4. save_image(): Saves each face image and associated labels to the image database.

Finally, the data is returned to the front-end to be displayed in the browser. If an error occurs at any point in the process, the function catches the error and returns a JSON response indicating that there was an error processing the image.

Database Design

Mongo DB was chosen for this project due to its ability to store, process and query large amounts of data faster and more efficiently than other database technologies such as an SQL database (IBM, MySQL vs. MongoDB: What's the difference, 2022) It is also free to use and open source. The project database has 2 collections: "Images.image_data" and "Images.Users".

A connection to the database is established in the backend server of the main application in the flaskapp.__init__.py file, and the label.py file of the labelling server. This allows the main application to connect to the database to save and update records and allows the labelling application to perform nearest neighbour search to correct label predictions.

The Users collection contains the information of registered users and is queried from the backend server to create, update, and delete user records. The collection has the following fields:

```
{_id: UUID
Name: String
password: String(SHA 256 encrypted)
admin: String (True or False) }
```

The Image_data collection stores all the information associated with each saved image and is queried from the backend server to create, update, and delete image records. The labelling server also queries this collection to perform nearest neighbour search. The collection has the following fields:

```
{_id: UUID
image_data: base64 encoded image string.
Embedding: Array
Unverified_labels: Array
Verified_labels: Array.
Incorrect_labels: Array
UserAddedLabels: Array
requiresVerification: String (True or False) }
```

Face Detection Server

The face detection server is a service that processes images and returns the coordinates of each face detected in the image, and cropped image of each face. To access the API, a POST request is made to the endpoint "http://127.0.0.1:5010/detect". The request must be in JSON and include the "img" key, which contains base64 encoded image data. An example request is shown below:

Python:

```python
with open("image.jpg", "rb") as image_file:
    image_data= base64.b64encode(image_file.read()).decode('utf-8')

image_data = req["img"]
endpoint = "http://127.0.0.1:5010/detect"
headers = {"Content-Type": "application/json"}

response = requests.post(endpoint, headers=headers, json={"img": image_data})
```

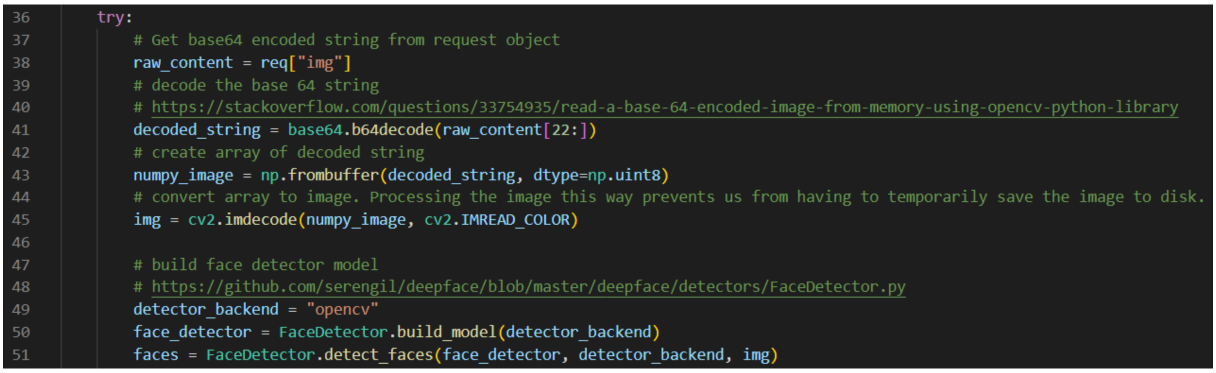
Figure 13: Example request in Python

JavaScript:

```javascript
const imageData = canvas.toDataURL( "image/png");
      fetch("http://127.0.0.1:5003//processImage", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          img: imageData,
        }),
      })
        .then((response) => response.json())
        .then((data) => {
          if(data.success !="True"){
            console.log(data.msg)
          }
}
```

Figure 14: Example request in JavaScript

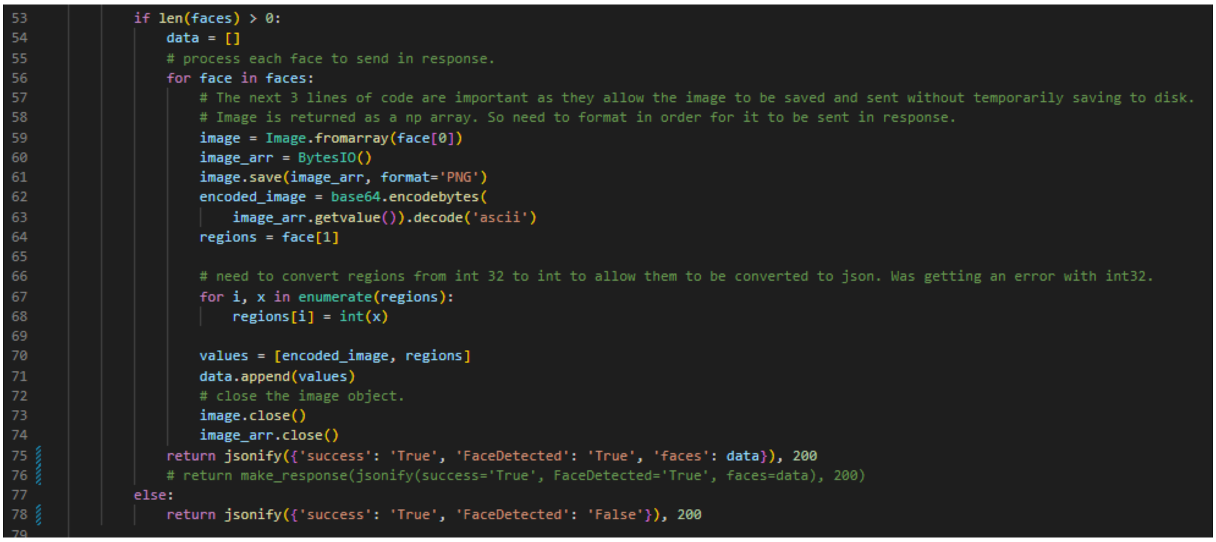
The endpoint expects a POST request and upon receiving one, the detect() function is called. This function first verifies if the request contains the "img" key. If image data is present in the request, it is decoded from a base64 encoded string to an image object using the NumPy and OpenCV libraries (see Figure 15). This allows the image to be efficiently processed without the need for temporary disk storage. The face_detector model is then built using the DeepFace.FaceDetectors module (Serengil et al., 2020). The model, the image, and the chosen detector backend are then passed as parameters into FaceDetector.detect_faces(). This method returns the number of faces detected in the image, the regional coordinates of each face detected in the image, and the image data of each cropped-out face.



```python
    try:
        # Get base64 encoded string from request object
        raw_content = req["img"]
        # decode the base 64 string
        # https://stackoverflow.com/questions/33754935/read-a-base-64-encoded-image-from-memory-using-opencv-python-library
        decoded_string = base64.b64decode(raw_content[22:])
        # create array of decoded string
        numpy_image = np.frombuffer(decoded_string, dtype=np.uint8)
        # convert array to image. Processing the image this way prevents us from having to temporarily save the image to disk.
        img = cv2.imdecode(numpy_image, cv2.IMREAD_COLOR)

        # build face detector model
        # https://github.com/serengil/deepface/blob/master/deepface/detectors/FaceDetector.py
        detector_backend = "opencv"
        face_detector = FaceDetector.build_model(detector_backend)
        faces = FaceDetector.detect_faces(face_detector, detector_backend, img)
```

Figure 15: Creating local image object and Face Detector Model

For each face detected by the Face Detector model, the face image is converted from a NumPy array to a Image object (lines 56-63 of Figure 16 below). The image data is then converted to a base64 encoded string using the encodeBytes() function. A peculiarity of the code here is that the regional coordinates for each face need to be converted from int32 to int to allow them to be converted to JSON, otherwise an error was being thrown.

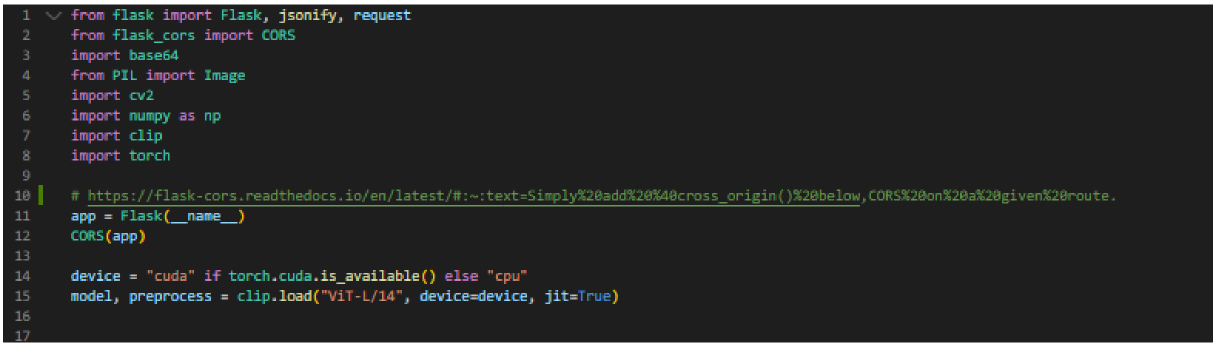


```python
53          if len(faces) > 0:
54              data = []
55              # process each face to send in response.
56              for face in faces:
57                  # The next 3 lines of code are important as they allow the image to be saved and sent without temporarily saving to disk.
58                  # Image is returned as a np array. So need to format in order for it to be sent in response.
59                  image = Image.fromarray(face[0])
60                  image_arr = BytesIO()
61                  image.save(image_arr, format='PNG')
62                  encoded_image = base64.encodebytes(
63                      image_arr.getvalue()).decode('ascii')
64                  regions = face[1]
65
66                  # need to convert regions from int 32 to int to allow them to be converted to json. Was getting an error with int32.
67                  for i, x in enumerate(regions):
68                      regions[i] = int(x)
69
70                  values = [encoded_image, regions]
71                  data.append(values)
72                  # close the image object.
73                  image.close()
74                  image_arr.close()
75              return jsonify({'success': 'True', 'FaceDetected': 'True', 'faces': data}), 200
76              # return make_response(jsonify(success='True', FaceDetected='True', faces=data), 200)
77          else:
78              return jsonify({'success': 'True', 'FaceDetected': 'False'}), 200
```

Figure 16: Formatting Image Data

Image Embedding Server

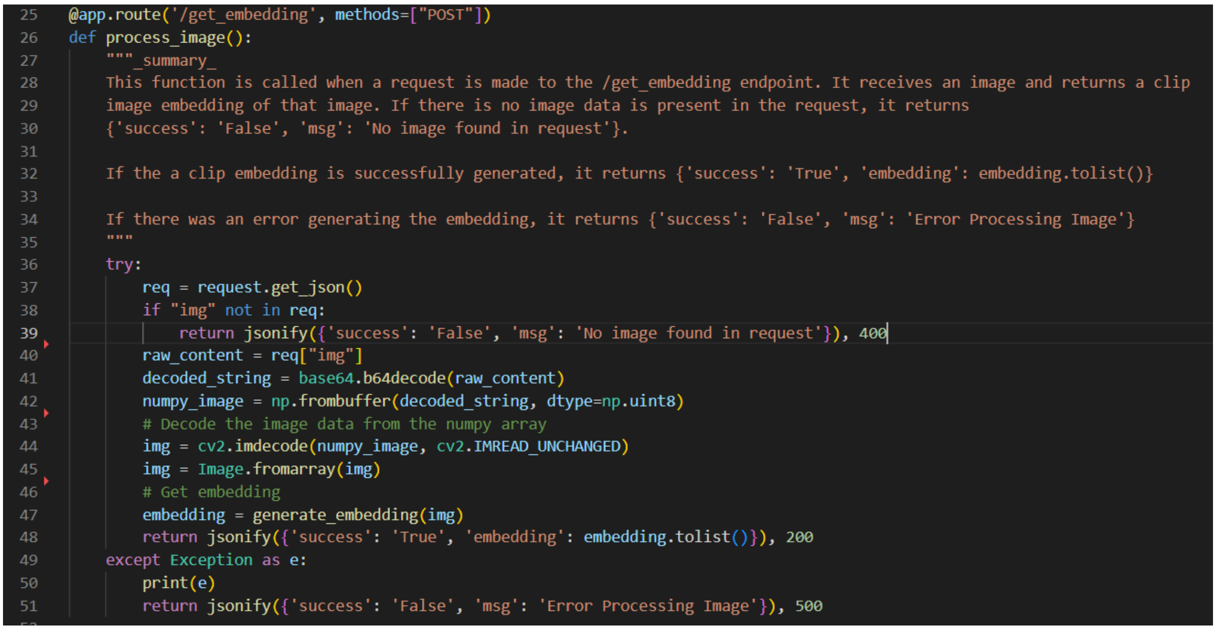
The image embedding server is a simple web service for generating CLIP image embeddings. On server startup, Cross-Origin Resource Sharing (CORS) support is enabled. This allows the web service to be called across different domains (lines 10-11 below). The image embedding model is then built using either the CPU or GPU depending on what is available on the machine the server is being run on.



```python
from flask import Flask, jsonify, request
from flask_cors import CORS
import base64
from PIL import Image
import cv2
import numpy as np
import clip
import torch

# https://flask-cors.readthedocs.io/en/latest/#:~:text=Simply%20add%20%40cross_origin()%20below,CORS%20on%20a%20given%20route.
app = Flask(__name__)
CORS(app)

device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-L/14", device=device, jit=True)
```

Figure 17: Initialising CLIP model

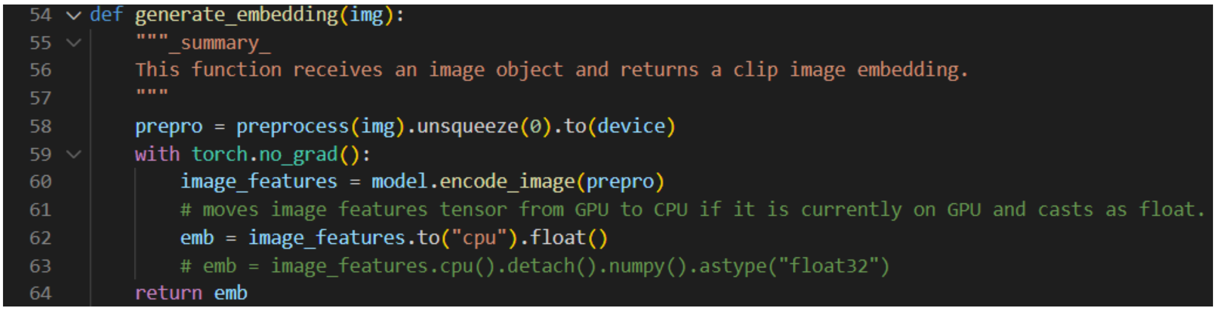
The ”/get_embedding” endpoint (shown below in Figure 18 below) expects a post request containing base64 encoded image data and the key “img”. If the "img" key is not present in the request, the endpoint will respond with the message "No image found in request". If image data exists in the request, it is decoded from a base64 string into an image object so that it can be processed by CLIP.



```python
@app.route('/get_embedding', methods=["POST"])
def process_image():
    """_summary_
    This function is called when a request is made to the /get_embedding endpoint. It receives an image and returns a clip
    image embedding of that image. If there is no image data is present in the request, it returns
    {'success': 'False', 'msg': 'No image found in request'}.

    If the a clip embedding is successfully generated, it returns {'success': 'True', 'embedding': embedding.tolist()}

    If there was an error generating the embedding, it returns {'success': 'False', 'msg': 'Error Processing Image'}
    """
    try:
        req = request.get_json()
        if "img" not in req:
            return jsonify({'success': 'False', 'msg': 'No image found in request'}), 400
        raw_content = req["img"]
        decoded_string = base64.b64decode(raw_content)
        numpy_image = np.frombuffer(decoded_string, dtype=np.uint8)
        # Decode the image data from the numpy array
        img = cv2.imdecode(numpy_image, cv2.IMREAD_UNCHANGED)
        img = Image.fromarray(img)
        # Get embedding
        embedding = generate_embedding(img)
        return jsonify({'success': 'True', 'embedding': embedding.tolist()}), 200
    except Exception as e:
        print(e)
        return jsonify({'success': 'False', 'msg': 'Error Processing Image'}), 500
```

Figure 18: ‘/get_embedding’ endpoint

The generate embedding() function takes an image as input and returns a CLIP image embedding. Figure 19 below shows the generate_embedding() function. First, the input image is processed into a format accepted by CLIP using the preprocess() function from the CLIP module. The pre-processed image data is then passed into the model.encode_image() function, which returns an image embedding.

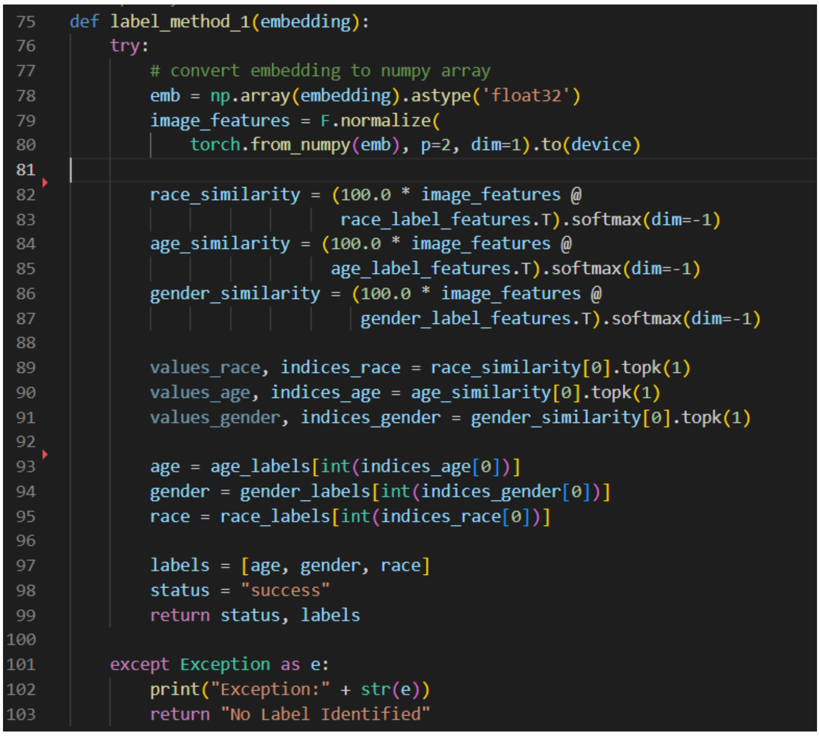


```
def generate_embedding(img):
    """_summary_
    This function receives an image object and returns a clip image embedding.
    """
    prepro = preprocess(img).unsqueeze(0).to(device)
    with torch.no_grad():
        image_features = model.encode_image(prepro)
        # moves image features tensor from GPU to CPU if it is currently on GPU and casts as float.
        emb = image_features.to("cpu").float()
        # emb = image_features.cpu().detach().numpy().astype("float32")
    return emb
```

Figure 19: generate_embedding()

Labelling Server

The labelling server is a Flask webs server with 4 endpoints: “/label_method_1”, “label_method_2”, “/label_method_3” and “/draw_labels”.[1] The first 3 endpoints represent 3 separate labelling methods which associate labels to an image. In Chapter 4 the accuracy of each method is measured and analysed. The “/label method _1” endpoint uses the clip module to match a text embedding to an image embedding. It does not have any self-correction method and is used as a baseline for experimentation. The endpoint receives a POST request containing an image embedding and passes the embedding to the label.label_method_1() function. The label_method_1() function receives the embedding and compares the similarity of the image embedding to the text embeddings that we wish to associate with the image. The text label associated with the closest matching text embedding for each of the label sets is then returned in the API response.



```
def label_method_1(embedding):
    try:
        # convert embedding to numpy array
        emb = np.array(embedding).astype('float32')
        image_features = F.normalize(
            torch.from_numpy(emb), p=2, dim=1).to(device)

        race_similarity = (100.0 * image_features @
                           race_label_features.T).softmax(dim=-1)
        age_similarity = (100.0 * image_features @
                          age_label_features.T).softmax(dim=-1)
        gender_similarity = (100.0 * image_features @
                             gender_label_features.T).softmax(dim=-1)

        values_race, indices_race = race_similarity[0].topk(1)
        values_age, indices_age = age_similarity[0].topk(1)
        values_gender, indices_gender = gender_similarity[0].topk(1)

        age = age_labels[int(indices_age[0])]
        gender = gender_labels[int(indices_gender[0])]
        race = race_labels[int(indices_race[0])]

        labels = [age, gender, race]
        status = "success"
        return status, labels

    except Exception as e:
        print("Exception:" + str(e))
        return "No Label Identified"
```

Figure 20: Label Method 1

[1] A extra endpoint “label_method_4” was added during the experimentation phase. This endpoint is a clone of “label_method_1”, however the prompts passed to clip have been improved. Please see Chapter 4: Experimentation, pg. 30.

The "/label_method_2" endpoint expects a post request containing a clip image embedding and returns the closest matching text labels. Labels returned by method_2 have been corrected by taking into account the incorrect labels of other similar images. The endpoint receives a POST request containing an image embedding and passes the embedding to the label.label_method_2() function (lines 160-229 of label_server/label.py). The label_method_2() function retrieves the three closest matching images from the database and identifies the labels that have been verified as incorrect for these images. These labels are used to correct the predicted labels for the current image. The idea behind this method is that if a predicted label is incorrect for another image with a similar embedding, it is assumed to be incorrect for the current image as well, and therefore not predicted. As the number of images in the database grows, the range of concepts and similarities between images should increase, making the system more accurate over time. A background scheduler is responsible for continually updating the index of images in the database, ensuring that the index is refreshed as the database grows (refer to lines 18-22 of label_server/app.py). Figure 21 below shows the flow of label_method_2:

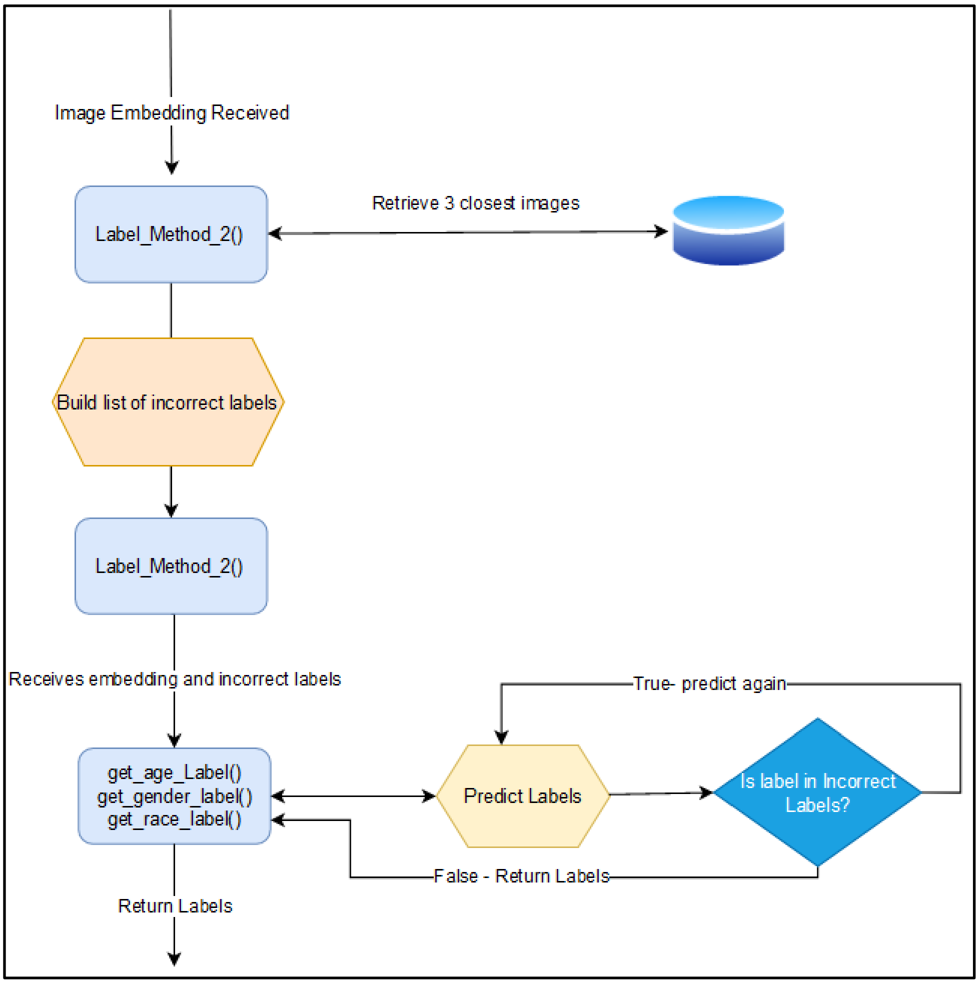
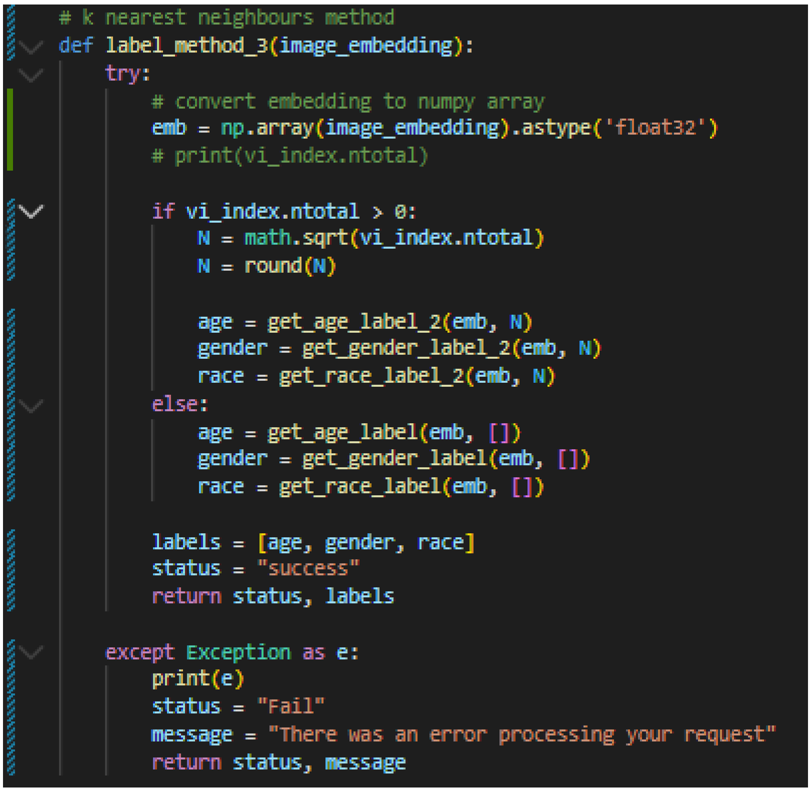


Figure 21: Label method 2 process flow

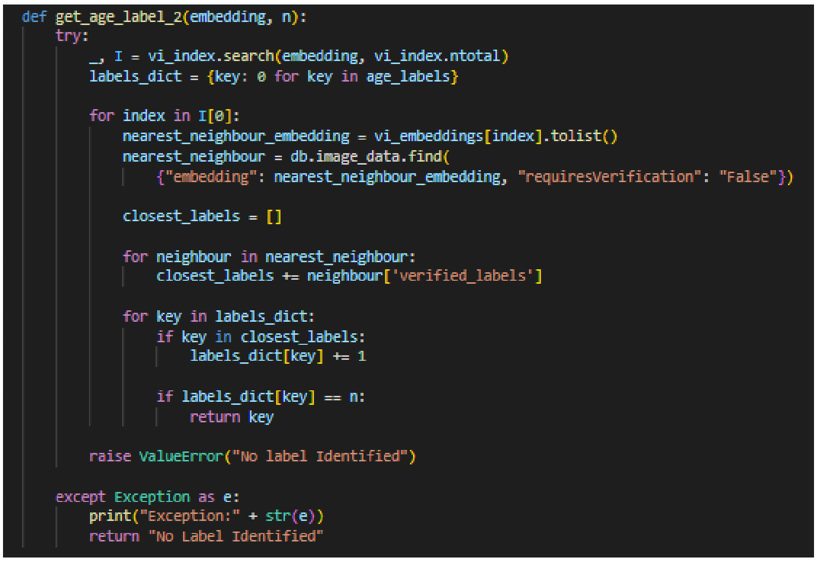
The “/label_method_3” endpoint uses k nearest neighbour search to predict the closest matching labels. The endpoint receives an image embedding and checks if the vi_index object contains data (this is a FAISS index of all verified images in the database instantiated at server startup and updated at regular intervals by a background scheduler). If this index does contain data, it calculates N as the square root of the length of the index, as shown below:



```
# k nearest neighbours method
def label_method_3(image_embedding):
    try:
        # convert embedding to numpy array
        emb = np.array(image_embedding).astype('float32')
        # print(vi_index.ntotal)

        if vi_index.ntotal > 0:
            N = math.sqrt(vi_index.ntotal)
            N = round(N)

            age = get_age_label_2(emb, N)
            gender = get_gender_label_2(emb, N)
            race = get_race_label_2(emb, N)
        else:
            age = get_age_label(emb, [])
            gender = get_gender_label(emb, [])
            race = get_race_label(emb, [])

        labels = [age, gender, race]
        status = "success"
        return status, labels

    except Exception as e:
        print(e)
        status = "Fail"
        message = "There was an error processing your request"
        return status, message
```

Figure 22: Label method 3

The embedding and N are then passed to get_age_label_2(), get_gender_label_2() and get_race_label_2(), each of which performs nearest neighbour search on the database to predict labels, as shown below:

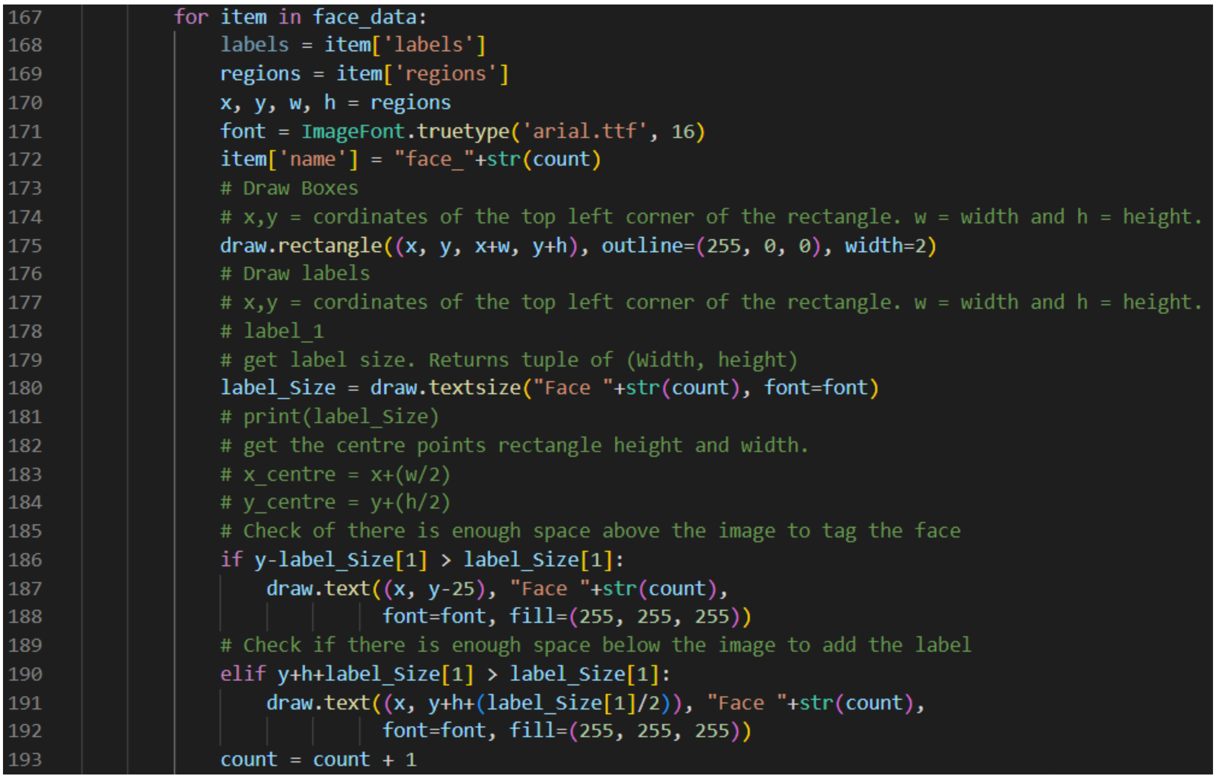


```
def get_age_label_2(embedding, n):
    try:
        _, I = vi_index.search(embedding, vi_index.ntotal)
        labels_dict = {key: 0 for key in age_labels}

        for index in I[0]:
            nearest_neighbour_embedding = vi_embeddings[index].tolist()
            nearest_neighbour = db.image_data.find(
                {"embedding": nearest_neighbour_embedding, "requiresVerification": "False"})

            closest_labels = []

            for neighbour in nearest_neighbour:
                closest_labels += neighbour['verified_labels']

            for key in labels_dict:
                if key in closest_labels:
                    labels_dict[key] += 1

                if labels_dict[key] == n:
                    return key

        raise ValueError("No label Identified")

    except Exception as e:
        print("Exception:" + str(e))
        return "No Label Identified"
```

Figure 23: get_age_label_2()

Once the nearest neighbours have been identified, the verified labels for that neighbour are retrieved from the image database. The first label to be verified as correct for a nearest neighbour N times is considered the correct label and returned.

Finally, the "/draw_labels" endpoint of the labelling server expects a POST request in JSON format that includes an image and the coordinates of any detected faces in the image. The endpoint returns the original image with a border drawn around each face. Initially, the plan was to draw each predicted label directly onto the original image taken by the user's webcam. However, this became impractical when multiple faces were detected, as taking into account the positions of each face and label, as well as the image boundaries, would lead to overlapping labels. To solve this problem, the code sequentially labels each face as "Face 1", "Face 2", and so on, making it easier for the user to associate each label with the corresponding face on the front end.



```
167         for item in face_data:
168             labels = item['labels']
169             regions = item['regions']
170             x, y, w, h = regions
171             font = ImageFont.truetype('arial.ttf', 16)
172             item['name'] = "face_"+str(count)
173             # Draw Boxes
174             # x,y = cordinates of the top left corner of the rectangle. w = width and h = height.
175             draw.rectangle((x, y, x+w, y+h), outline=(255, 0, 0), width=2)
176             # Draw labels
177             # x,y = cordinates of the top left corner of the rectangle. w = width and h = height.
178             # label_1
179             # get label size. Returns tuple of (Width, height)
180             label_Size = draw.textsize("Face "+str(count), font=font)
181             # print(label_Size)
182             # get the centre points rectangle height and width.
183             # x_centre = x+(w/2)
184             # y_centre = y+(h/2)
185             # Check of there is enough space above the image to tag the face
186             if y-label_Size[1] > label_Size[1]:
187                 draw.text((x, y-25), "Face "+str(count),
188                           font=font, fill=(255, 255, 255))
189             # Check if there is enough space below the image to add the label
190             elif y+h+label_Size[1] > label_Size[1]:
191                 draw.text((x, y+h+(label_Size[1]/2)), "Face "+str(count),
192                           font=font, fill=(255, 255, 255))
193             count = count + 1
```

Figure 24: Bordering each face detected in the image

Line 175 of Figure 24 above draws a border around the image based on the regional coordinates received in the request. Line 180 calculates the size of the label that will be used to tag the face and line 186 checks if there is sufficient space above the face border to draw the label. If there is enough space available, the label is drawn above the border. Else, the code checks if there is enough space below the border to draw the label. If space is available below the border, the label is drawn below the face border. This dynamic repositioning of the label ensures that the entire label is visible to the user without getting cut off.

Environment Setup

To set up the environment for running the server-side architecture, follow these steps:

1. Download and install Python version 3.9.
2. Install the virtual environment package by running the following command in your terminal: "pip install virtualenv".
3. Create a virtual environment with Python 3.9 by running the following command: "python -m virtualenv -p=<path_to_your_python3.9_executable>.venv".
4. Activate the virtual environment by navigating to the folder containing the .venv folder created in step 3 and running the command ". venv/scripts/activate".

With the virtual environment activated, install the required pip packages by running the following commands:

- "pip install Flask"
- "pip install Flask-Cors"
- Install PyTorch version 1.7.1 or later. Version 1.7.1 is recommended as later versions may cause unexpected errors. Follow the instructions at https://pytorch.org/ to install the recommended package for your machine.
- "pip install ftfy regex tqdm"
- "pip install git+https://github.com/openai/CLIP.git"
- "pip install deepface"
- "pip install faiss"
- "pip install numpy"
- “pip install apscheduler”
- “pip install pymongo”
- “pip install passlib”

Your environment should now be set up and ready to run the server-side architecture of this project.

Chapter 4: Experimentation

Manually experimenting and testing this system at scale would be extremely time-consuming and require a significant number of volunteers from a diverse pool of race and age groups. Fortunately, there are pre-existing labelled datasets that are publicly available, which can simulate having a large group of diverse test subjects. Two such datasets are Celeb-A (Liu, Luo, Wang, and Tang 2015) and FairFace (Karkkainen, Kimmo, and Joo, Jungseock, 2021).

To measure the effectiveness and accuracy of the system, we can pass an image embedding to the label server and compare the returned labels against verified labels from the dataset. For this experiment, we chose the FairFace dataset over Celeb-A for two reasons. Firstly, FairFace contains race, age, and gender-balanced images from 7 racial groups (White, Black, Indian, East Asian, Southeast Asian, Middle Eastern, Latino) and 9 age groups (0-2, 3-9, 10-19, 20-29, 30-39, 40-49, 50-59, 60-69, and more than 70). Secondly, the images in the FairFace dataset are pre-cropped and closely resemble the faces detected by the system via the user's webcam.

25: Image from FairFace dataset

26: Image detected by the system from the user's webcam.

Clip embeddings were generated for images from the FairFace dataset and saved for later use (see scripts/getEmbeddings.ipynb and Appendices, Scripts, getEmbeddings.ipynb). To ensure easy identification, the image name was added to the embedding filename. The full folder containing all 40,000 embeddings can be found in the 'Clip_Image_Embeddings/FairFace' folder. Next, 4000 embeddings were randomly selected and split into 20 folders, each containing 200 embeddings (see 'scripts/Test_File_Setup.ipynb' and 'Appendices, Scripts, Test File Setup Script').
Five rounds of testing were conducted for each of the labelling methods discussed in Chapter 3. Before each round the database was cleared of all data and the order of the test data was randomised. The experimentation script can be found at 'scripts/Experimentation_Script.ipynb'. Results can be found in the 'Results' folder.

Method 1 Results (Clip Only Method):

The Clip Only labelling method, which uses clip to match a clip text embedding of the exact label to a clip image embedding, had an average accuracy of 48.2%. The average number of correct predictions across the 5 folds ranged from 280 to 296, while the average number of incorrect predictions ranged from 304 to 320. The average number of correct predictions was 289.25, while the average number of incorrect predictions was 310.75.

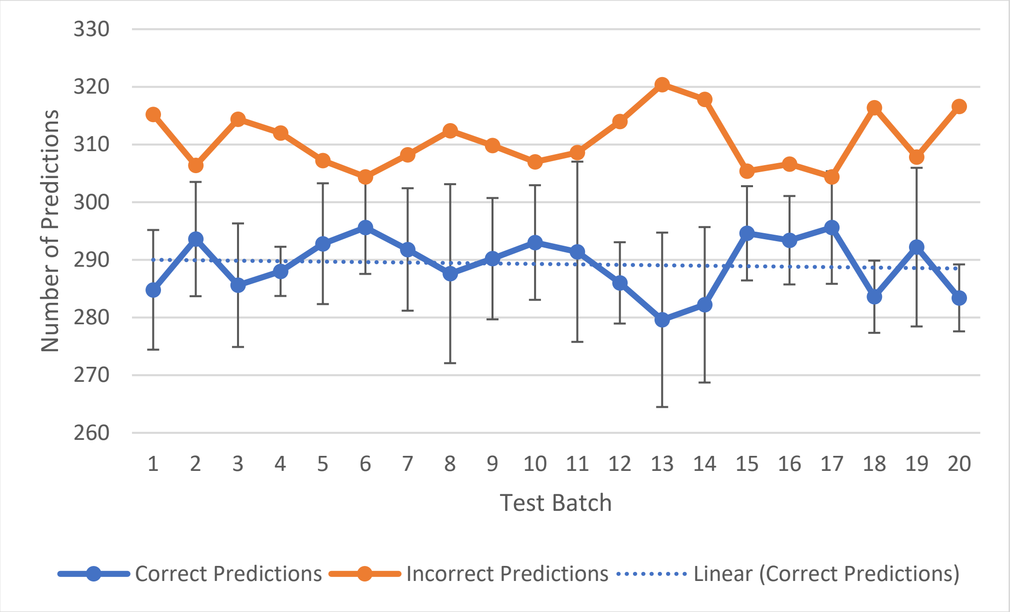


Figure 27: Label method 1 results

Overall, there is no general trend that can be identified from the test carried out, with the accuracy remaining stable and not getting better or worse as more images were added to the database. However, this is expected as this method of labelling has no correction mechanism. Full result files for method 1 testing can be found in the 'Results/Method 1 -Results' folder.

Since the performance of CLIP is known to be enhanced by providing more context to the label (Radford et al., 2021), I experimented further to explore how varying the prompts given to CLIP could improve its results. To achieve this, I created the sentence_builder() function (see lines 27-49 of label.py and Appendices, Functions, sentence_builder()), which takes in the list of FairFace labels and generates all possible label combinations in the following structure:

"A photo of a 'age_Label' year old 'race_Label' 'gender_Label'."

This produced a list of all possible age, race, and gender label combinations, as shown in the example below:

```
['A photo of a 0-2 year old White Male.',
 'A photo of a 0-2 year old White Female.',
 'A photo of a 0-2 year old Black Male.',
 'A photo of a 0-2 year old Black Female… etc.
```

I then cloned the method 1 endpoint into a new endpoint called 'label_method_4', which uses the new list of prompts to label each image. The results are shown below in Figure 28 below:

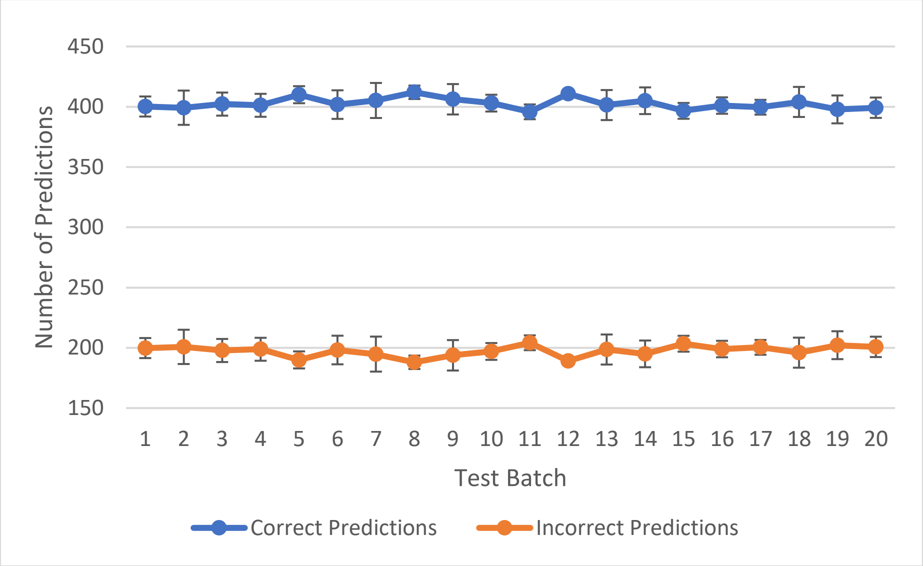


Figure 28: Results with improved prompt

As can be seen from the graph above, this approach produced a much higher level of accuracy, with an average accuracy of 67% across 5 folds of testing. The average number of correct predictions was 402.7 while the average number of incorrect predictions was 197.3. The accuracy of predictions remained relatively constant, which was expected as this method has no mechanism for correcting predictions. Full result files can be found in the 'Results/ Method 4 - Clip Improved Prompt' folder.

Method 2 Results (Don't Be Wrong Method):

For method 2, the average number of correct predictions across the 5 folds of testing was 288.39 (ranging between 279.4 and 297.8), while the average number of incorrect predictions was 311.61 (ranging between 302.2 and 320.6). The average accuracy was slightly less than that of method 1, at 48.06%. However, the results do appear to show correct predictions trending slowly upward.

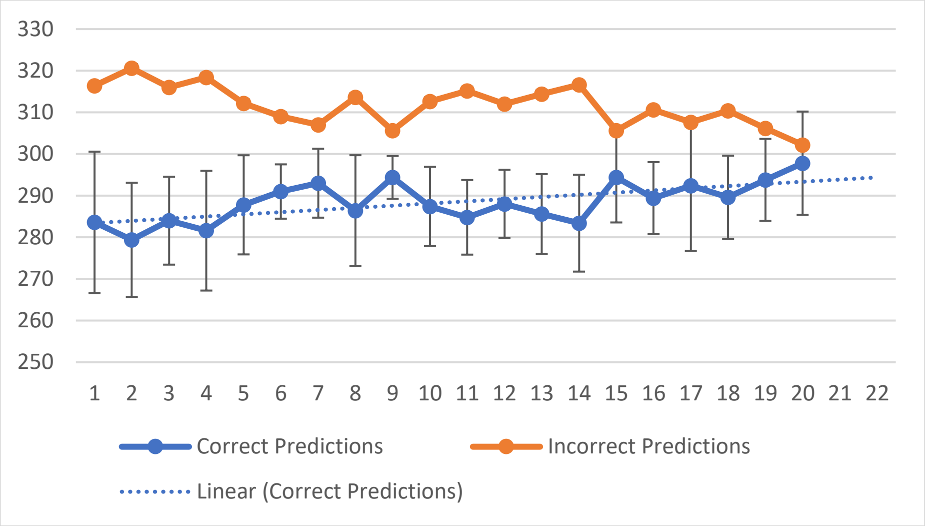


Figure 29: Method 2 results

To see if this upward trend continued, the experiment was repeated with double the number of test image embeddings (40 folders containing 200 embeddings each as opposed to 20 folders). As can be seen from Figure 30 below, no meaningful improvement in accuracy was recorded after batch 12, with an average accuracy of 48.8%. I can therefore conclude that method 2 does not lead to any meaningful improvements in accuracy over time. Full result files can be found in the 'Results/ Method 2 – Results' and 'Results/Method 2 – Retest Results' folders.

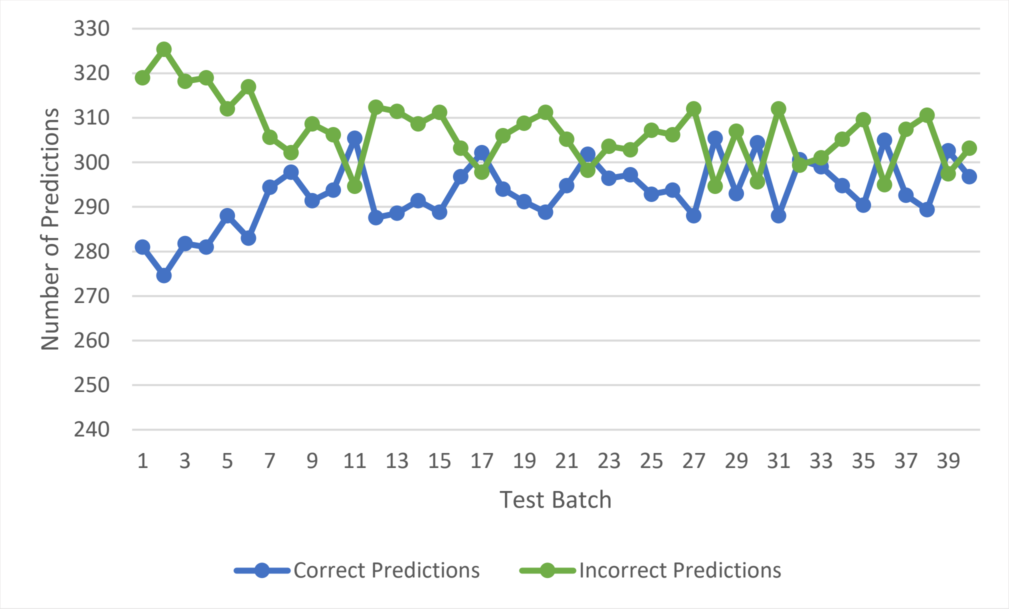


Figure 30: Method 2 Retest

Method 3 Results (K Nearest Neighbour):

Labelling method 3 had an average accuracy of 66.13% over five folds of testing. The average number of correct predictions across the 5 folds of testing was 396.76 (ranging between 417 and 305), while the average number of incorrect predictions was 203.24(ranging between 183 and 295).

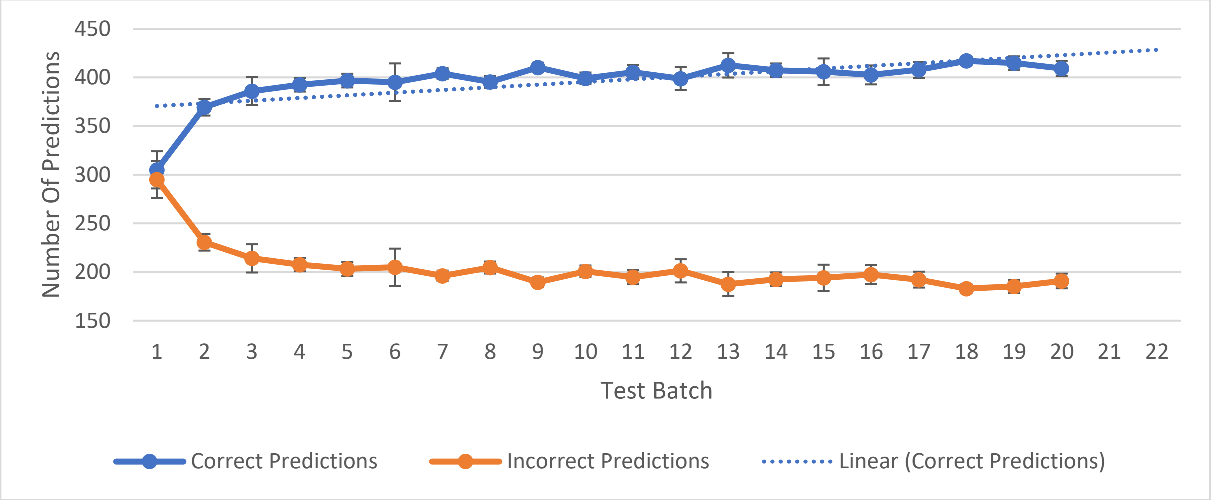


Figure 31: Method 3 - Knn results

Accuracy trended positively upwards, from a low of 50.8% at batch 1 to a high of 69.5% at batch 18. Batches 19 and 20 showed a slight decrease in accuracy, however, to 69.13% and 68.2%. Full result files can be found in the 'Results/ Method 3 - NearestNeighbour Results'.

Based on the results above, it can be concluded that method 3 (K nearest neighbour) was the most accurate among the three original labelling methods, and it was the only method that showed improvement in accuracy over time. However, after modifying the prompts given to CLIP in method 1 (CLIP only), its average level of accuracy improved significantly and reached the highest level at 67.2%. Moreover, method 1 with a modified prompt was faster than method 3, despite its superior performance. Therefore, if I were to continue development and experimentation, I would focus my efforts on prompt engineering to measure how much accuracy can be improved with different prompt structures, and how these prompts can be improves over time.

Chapter 5: Testing

Requirements testing

Images have been removed from the table below to meet space requirements. However images for tests carried out can be seen in the 'test_plan.xlsx' file, under the 'Req Testing – Images' tab.

Req	AC	Test Description	Pass/Fail	Notes	Fix	Retest
1	AC1: When a user visits the designated web address (http://localhost:5000/analyser), they will be prompted to grant permission for the system to access their webcam.	Navigated to the URL at http://localhost:5000/analyser.	PASS	When I navigate to the URL at http://localhost:5000/analyse, the pop-up below is shown.	N/A	N/A
	AC2.If the user grants permission, the system will display a live video feed of the webcam stream on the screen. There will be a button for each labelling method There will be a "Stop" button which stops any labelling process in progress.	Navigated to the URL at http://localhost:5000/analyser and clicked "Allow" on the webcam permission pop-up (See test 1)	PASS	Webcam access was granted, and a live stream was shown on screen. Two buttons as described in the acceptance criteria are displayed below the live stream. See the image below:	N/A	N/A
	AC3. The page should be available to registered and anonymous users.	Navigated to the URL at http://localhost:5000/analyser as an anonymous user, a registered user and an admin user	PASS	Access granted to the page for all 3 role types.	N/A	N/A
2	AC1. Upon clicking on any of the labelling buttons, the system should initiate the labelling process for that labelling method	Clicked on each of the "Method 1", Method 2", and "Method 3" buttons.	PASS	After clicking on each button, there was a brief delay, and a labelled image was displayed on the screen. After another brief delay, a second image was displayed on the screen without me needing to press the button again. This continued until I pressed the "Stop" button. When I clicked on the button initially, all other buttons were disabled, except for the "Stop" button.	N/A	N/A

	AC 3. The user should only need to press a labelling method button once, which should start the labelling process. In case of any error, the system should automatically attempt to restart the process.	See test 4 results above	PASS	See test 4 results above	N/A	N/A
	AC4.When the labelling process is in progress, all labelling method buttons should be disabled.	I clicked on each of the method buttons. This took 3 screenshots and began the image labelling process 3 times, causing the system to slow.	FAIL	This took 3 screenshots and began the image labelling process 3 times, causing the system to slow	Disabled button once it is clicked. It is only re-enabled once the user clicks the "Stop" button.	PASS
	AC5. The "Stop" button should halt any process that is in progress and re-enable the labelling method buttons.	Clicked on the stop button when the image labelling process was in progress	PASS	The current image labelling process finished, and the process stopped	N/A	N/A
3	AC1. When a face is detected in an image, the system should crop the face and save the image for further processing (i.e., generating an image embedding).	Sent a postman request to the Face_Detect_Server containing base 64 image data representing an image containing a human face.	PASS	The API returned a response stating that faces were detected, Base 64 image data of each face, and the regional coordinates of each face detected in the image	N/A	N/A
	AC2. In case a face is not detected in the image, the system should continue to sample frames from the webcam and repeat the face detection process until a face is detected.	Clicked on each labelling method button when no face was in the webcam.	PASS	The message "No Face Detected" was displayed to the console and the process repeated.	N/A	N/A
4	AC1.For each face detected in the image, the face is cropped and saved for processing (generating an image embedding).	Sent a postman request to the Face_Detect_Server containing base 64 image data representing an image containing a human face.	PASS	The API returned a response stating that faces were detected, Base 64 image data of each face, and the regional coordinates of each face detected in the image	N/A	N/A
	AC2. If no face is detected in the image, the system should continue to sample frames from the user's webcam and repeat the process until a face is detected.	Clicked on each labelling method button when no face was in the webcam.	PASS	The message "No Face Detected" was displayed to the console and the process repeated.	N/A	N/A
5	AC1.For each face detected, a red,	Clicked on each labelleing	PASS	The image was returned with a red border	N/A	N/A

	rectangular border should be drawn around the face in the original image.	method button to start the image analyses process and waited for the labelled image to be returned.		drawn around the face in the image and labelled "Face 1".		
	AC2.Each face should be labelled.	Clicked on each labelling method button to start the image analyses process and waited for the labelled image to be returned.	PASS	See test 12 above.	N/A	N/A
	AC3. Labels applied to faces should not be cut off by image borders or other faces.	Clicked on each labelling method button to start the image analyses process and waited for the labelled image to be returned. Moved my face to the four borders of the webcam and waited for the system to return the results.	PASS	Each labelled image returned had the face label visible on the image. If there was no room to draw the label above the image, the label was drawn below the image. Else the label was drawn above the image.	N/A	N/A
6	AC1. The system can generate a clip embedding for each face detected in an image captured by a user's webcam.	Sent a postman request to the image_embedding server containing base 64 image data.	PASS	Received a response from the image_embedding server containing a clip embedding for the image. (See Postman Test files).	N/A	N/A
7	AC1.When an image embedding is generated, the system can match the closest text embedding from a set of labels to that image.	Sent a postman request to the labelling server containing an image embedding.	PASS	Received an API response containing the labels matched to the embedding (See Postman Test files).	N/A	N/A
8	AC1. When a labelled image has been received, it should be displayed to the user.	Clicked on each labelling method button and waited for the API to respond.	PASS	A cropped-out image of the face with labels is displayed on the screen.	N/A	N/A
	AC2.The original image should also be displayed to the user, with borders drawn around each detected face.	Clicked on each labelling method button and waited for the API to respond.	PASS	The original screenshot was returned and displayed on the screen with faces bordered.	N/A	N/A
9	1. There should be at least 1 method of correcting incorrect label predictions.	See Chapter 4 Experimentation. Label Method 3 proved to become accurate over time	🟩	See Chapter 4 Experimentation. Label Method 3 proved to become accurate over time	N/A	N/A

	2. The accuracy of any label correction methods developed should be measured and recorded in Chapter 4.	See Chapter 4 Experimentation.		See Chapter 4 Experimentation.	N/A	N/A
10	AC1.When a labelled image is returned to the user, the process automatically begins again.	Clicked on each labelling method button and waited for the API to respond.	PASS	Once a labelled image was returned, another screenshot was taken, and the process started automatically without any input being required.	N/A	N/A
	AC2.The process should only stop when the user exits the webpage or presses the "Stop" button (see Requirement 2).	See test 7	PASS	See test 7	N/A	N/A
11	AC1.The screen should only be accessible to registered users. If a non-registered user tried to access the screen by entering the URL directly into the browser, they should be navigated away from the screen.	Navigated to the screen at http://localhost:5000/verify/ as a registered user. Then I navigated to the screen at http://localhost:5000/verify/ as an anonymous user.	PASS	As a registered user, I was able to access the page and its functionalities. As an anonymous user, I was redirected to the home page.	N/A	N/A
	AC2.The screen should retrieve the next unverified image from the database. On the left of the screen the image should be displayed. On the right of the screen, all unverified labels should be displayed.	Navigated to the screen at http://localhost:5000/verify/ as a registered user.	PASS	As a registered user, when I navigated to the screen and was shown an image on the left-hand side and a set of labels on the right-hand side. I checked the image ID in the database and this image was an unverified image.	N/A	N/A
	AC3.There should be a check box for each label.	Navigated to the screen at http://localhost:5000/verify/ as a registered user.	PASS	As a registered user, when I navigated to the screen there was a set of labels on the right-hand side, with a check box beside each label. The checkbox was active and could be toggled.	N/A	N/A
	AC4.There should be a "Confirm Labels" button below the labels. When the user clicks this button, the user's selection should be saved, and the user should be shown a screen that allows them to add their own labels to the image	Navigated to the screen at http://localhost:5000/verify/ as a registered user. Checked the top checkbox and clicked "Confirm Labels"	PASS	As a registered user, when I checked the top checkbox and clicked "Confirm Labels", I was shown the next screen in the form (user added labels). When I clicked the back button, my selections remain selected.	N/A	N/A

12	AC1. After verifying the labels generated by the system, a registered user should be able to add additional labels to the image.	Performed tests for Req 11 above and was shown a screen that allowed me to add my own labels.	PASS	Attempted to add a label and the label was shown on the screen. Continued to add labels until a scrollable area was triggered. The system allowed me to add as many labels as I wanted.	N/A	N/A
	AC2.Users can add 0 or more labels to an image as they see fit.	Attempted to add multiple labels	PASS	I was able to continue to add as many labels as I wanted to.	N/A	N/A
	AC3.Users should be able to remove any additional labels they have attached before submitting the image. (I.e., in case of spelling mistakes or if they have reconsidered the label).	Clicked on the trash icon beside each label.	PASS	Each label was removed from the list until no labels were remaining.	N/A	N/A
	AC4. The user should not be allowed to add empty labels.	Attempted to add a label with only white space.	FAIL	I was able to add an empty label:	Added label validation	PASS
	AC5. There should be a submit button that allows the user to submit additional labels to the database. The record for the image should be updated with the field "UserAddedLabels" that contains the labels added by the user.	Added user labels and clicked submit	FAIL	The verified labels and the user-added labels were passed through to the backend server. However, only the user-added labels were not saved to the database.	Added code to add the field "UserAddedLabels" when the record is updated. As this feature was developed later, the original update logic did not include user-added labels. Updated update_labels() to now process and save user-added labels. When the test was rerun, all data was submitted to the database as expected.	PASS
13	AC1.When the user enters a recognised username and password and clicks submit, they are logged into the system and navigated to the Home Page page.	Navigated to the screen http://127.0.0.1:5000/login/. Logged in using the following details: username: user1 password: 123qwe	PASS	Upon entering valid details into the login form and clicking on the "Log In" button, I was navigated to the home page.	N/A	N/A
	AC2.When the user enters incorrect details and clicks submit, an error message saying "Invalid Credentials"	Navigated to the screen http://127.0.0.1:5000/login/. Logged in using the following	FAIL	Incorrect Error message shown. See the image below. Need to fix the spelling.	Spelling Corrected	PASS

	should be displayed.	details: username: incorrect_Username password: incorrect_password	🟥			🟩
14	AC1.Admin users should have access to all pages on the application and have access to all functionalities.	Logged into the application as an admin user and attempted to navigate to all pages in the application as below : - Create Account -Login -Home -Face Analyser -Image Verification	PASS	As an admin user, I was able to access all screens on the application.	N/A	N/A
	AC2.Registered users should have access to the “Analyse”, “Verify” and “Login” pages. They should be able to use the image analyser and also verify labels that have been attached to images on the verify page.	Logged into the application as an admin user and attempted to navigate to all pages in the application as below : - Create Account -Login -Home -Face Analyser -Image Verification	PASS	As a registered user, I was able to access the Login, Home, Face Analyser, and Image Verification Pages. When I attempted to navigate to the create account page via URL, I was redirected to the home screen.	N/A	N/A
	AC3. Anonymous users should have access to the “Analyse” and “Login” pages and functionality.	Logged into the application as an admin user and attempted to navigate to all pages in the application as below : - Create Account -Login -Home -Face Analyser -Image Verification	PASS	As an anonymous user able to access the home, login, and Face Analyser pages. I was redirected to the home screen when I attempted to navigate to the verification and create account pages via URL.	N/A	N/A

15	AC1.The “Account Creation” page should be accessible to admin users only.	Pass - See Test 27	PASS	Pass - See Test 27	N/A	N/A
	AC2.There should be a form containing the fields “Username” and “Password”. There should also be a checkbox labelled “Is Admin”.	Logged into the application as an admin user and navigated to the account creation page.	PASS	The Create Account page shows a form containing a username field, a password field, and an isAdmin checkbox.	N/A	N/A
	AC3.Username and password cannot be blank or greater than 50 characters.	Attempted to create a username with greater than 50 characters	FAIL	Fail - I was able to create a username 51 characters long.	Validation added to validate username not greater than 50 characters. 49chars - pass 50chars - pass 51 chars- fail	PASS
	AC4.When the user clicks the “Submit” button, the new user details should be validated. If the credentials are valid, a new user should be created.	Attempted to create a new user using valid details and clicked the submit button	PASS	The user details were validated and successfully saved to the users' database. I then was able to successfully log into the system using the details.	N/A	N/A

	AC5.If the details are invalid, an appropriate error message should be displayed (i.e. “Password cannot be blank”).	Attempted to create a new user with invalid details (username and password >50 chars) and a user with no password.	FAIL	When I attempted to create a user with no password, no user was created but the form did not show an appropriate error message. When I attempted to create a user with a password >50 chars and a username >50 chars, no error was thrown, and the user was created.	Validating updated. The error message now shows when the user enters an empty username or password. Validation was also added to validate that the username and password are <= 50 characters.	PASS
	AC6.If the new user details are valid but a user with the same username already exists on the system, the message “User already exists with this username” should be displayed.	Attempted to create a duplicate user	PASS	When attempting to create a duplicate user, the following error message was shown : "Username already exists. Please Choose Another"	N/A	N/A

Exploratory Testing – Front-End Application

Test	Bug Description	Pass/Fail	Fix	Retest
Browser Testing: Chrome	Ran through each page and each functionality on Google Chrome. No abnormal behaviour was detected.	Pass	N/A	N/A
Browser Testing: Firefox	Ran through each page and each functionality on Firefox. No abnormal behaviour was detected.	Pass	N/A	N/A
Browser Testing: Edge	Ran through each page and each functionality on Firefox. No abnormal behaviour was detected.	Pass	N/A	N/A
Login Page - Attempted to break and find errors on the page.	No abnormal behaviour detected	Pass	N/A	N/A

Create Account Page - Attempted to break and find errors on the page.	Bug - Enter a username greater than 50 characters and a valid password and click submit. A validation message appears and the user is not created. Repeat this step but this time check the "User is admin" checkbox. A user is created.	Fail	Field validations were not applied to the '/user/create_admin' route. To solve this, the correct validations were applied.	Pass
Image Verification Page - Attempted to break and find errors on the page.	No abnormal behaviour detected	Pass	N/A	N/A
Home Page - Attempted to break and find errors on the page.	No abnormal behaviour detected	Pass	N/A	N/A

Exploratory Testing – Application Backend Server

Test	Result	Pass/Fail	Fix	Retest
Send a request to the backend server "/processimage" endpoint with no img key or image data.	Response: {"msg": "No image found in request", "success": "false"}	Pass	N/A	N/A

Send a request to the backend server "/processimage" endpoint with the img key but no image data.	Response:
	{
	"msg": "There was an error processing the image",
	"success": "false"
	}
		Pass	N/A	N/A

Send a request to the backend server "/processimage" endpoint with the img key but invalid (corrupted) data.	Response:
	{
	"msg": "There was an error processing the image",
	"success": "false"
	}
		Pass	N/A	N/A

Send a request to the backend server "/processimage" endpoint with img and valid image data, but the image has no face.	Response:
	{
	"success": "True",
	"FaceDetected": "False"
	}
		Pass	N/A	N/A

Send a request to the backend server "/processimage" endpoint with img and valid image data that contains a single face.	Response:
	{
	"FaceDetected": "True",
	"face_data": [
	{
	"embedding": [
	[
	0.39716845750808……,
	]
	]
	"image": Bwl/AAEAAElEQVR4nOz9SZMly5leiOlgZj6c….",
	"success": "True"
	}
		Pass	N/A	N/A

Exploratory Testing – Face Detection Server

Test	Result	Pass/Fail	Fix	Retest

Send a request to the Face detect server with no img key or image data.	{
	"error": "No Image Detected", "success": "False"
	}
		Pass	N/A	N/A

Send a request to the Face Detect server with an img key but no image data.	{
	"error": "Error processing image", "success": "False"
	}
		Pass	N/A	N/A

Send a request to the Face detect server with an img key but no invalid (corrupted) data.	{
	"error": "Error processing image", "success": "False"
	}
		Pass	N/A	N/A

Send a request to the Face detect server with the img key and image data but no face in the image.	Response:
	{
	"success": "True", "FaceDetected": "False"
	}
		Pass	N/A	N/A

Send a request to the Face Detect server with the img key and image data with a face in the image.	{
	"FaceDetected": "True",
	"faces": [
	[
	"iVBORw0KGgoAAAANSUhEUgAAATEAAAExCAIAAAC......,
	[
	139,
	88,
	305,
	305
	]
	]
	],
	"success": "True"
	}
		Pass	N/A	N/A

Exploratory Testing – Image Embedding Server

Test	Result	Pass/Fail	Fix	Retest

Send a request to the image embedding server with no img key or image data.	{
	"msg": "No image found in request",
	"success": "False"
	}
		Pass	N/A	N/A

Send a request to the image embedding server with an img key but no image data.	{
	"msg": "Error Processing Image",
	"success": "False"
	}
		Pass	N/A	N/A

Send a request to the image embedding server with an img key but no invalid (corrupted) data.	{
	"msg": "Error Processing Image",
	"success": "False"
	}
		Pass	N/A	N/A

Send a request to the image embedding server with the img key and valid image data	{
	"embedding": [
	[
	0.4766950011253357,
	-0.09165439009666443,
	0.2264534085................,
	0.8860663175582886
	]
	],
	"success": "True"
	}
		Pass	N/A	N/A

Exploratory Testing – Labelling Server

Test	Result	Pass/Fail	Fix	Retest

Send a request to the '/label_method_1'endpoint with an embedding key or but no data.	Response: {
	"msg": "Error in retrieving labels",
	"success": "False"
	}
		Pass	N/A	N/A

Send a request to the '/label_method_1' endpoint with no embedding key or data.	c	Pass	N/A	N/A
Send a request to the '/label_method_1' endpoint with the embedding key and data.	Response: { "labels": ["40-49", "Male", "White"],	Pass	N/A	N/A

	"success": "True" }			

Send a request to the '/label_method_2' endpoint with an embedding key or but no data.	Response: {
	"msg": "Error in retrieving labels",
	"success": "False"
	}
		Pass	N/A	N/A

Send a request to the '/label_method_2' endpoint with no embedding key or data.	Response: {
	"msg": "Error in retrieving labels",
	"success": "False"
	}
		Pass	N/A	N/A

Send a request to '/label_method_2' endpoint with the embedding key and data.	Response:
	{
	"labels": [
	"40-49",
	"Male",
	"White"
	],
	"success": "True"
	}
		Pass	N/A	N/A

Send a request to the '/label_method_3' endpoint with an embedding key but no data.	Response: {
	"msg": "Error in retrieving labels",
	"success": "False"
	}
		Pass	N/A	N/A

Send a request to the '/label_method_3' endpoint with no embedding key or data.	Response: {
	"msg": "Error in retrieving labels",
	"success": "False"
	}
		Pass	N/A	N/A

Send a request to '/label_method_3' endpoint with the embedding key and data.	Response:
	{
	"labels": [
	"40-49",
	"Male",
	"White"
	],
	"success": "True"
	}
		Pass	N/A	N/A

Send a request to label the servers '/draw_labels' endpoint with no img key, image data, or facial regions.	Status: 500INTERNAL SERVER ERROR returned.	Fail	Added error handling validation to the endpoint. Response below.
			{
			"error": "An error occurred: 'img'",
			"success": false
			}
				Pass

Send a request to label the servers '/draw_labels' endpoint with the img key but no data.	Status: 500INTERNAL SERVER ERROR returned.	Fail	Added error handling validation to the endpoint. Response below.
			{
			"error": "No image data provided",
			"success": false
			}
				Pass

Send a request to label the servers '/draw_labels' endpoint with the img key and invalid image data.	Status: 500INTERNAL SERVER ERROR returned.	Fail	Added error handling validation to the endpoint. Response below.
			{
			"error": "An error occurred: cannot identify image file",
			"success": false
			}
				Pass

Send a request to label the servers '/draw_labels' endpoint with no 'face_data' key	Status: 500INTERNAL SERVER ERROR returned.	Fail	Added error handling validation to the endpoint. Response below.
			{
			"error": "An error occurred: 'face_data'",
			"success": false
			}
				Pass

Send a request to label servers ‘/draw_labels’ endpoint with the 'face_data' key but no data	Status: 500INTERNAL SERVER ERROR returned.	Fail	Added error handling validation to the endpoint. Response below.
			{
			"error": "No face data provided",
			"success": false
			}
				Pass

Send a valid request to the ‘/draw_labels’ endpoint	Sent a valid request containing image data and a valid face data object. Received an image with the face bordered.	Pass	N/A	N/A

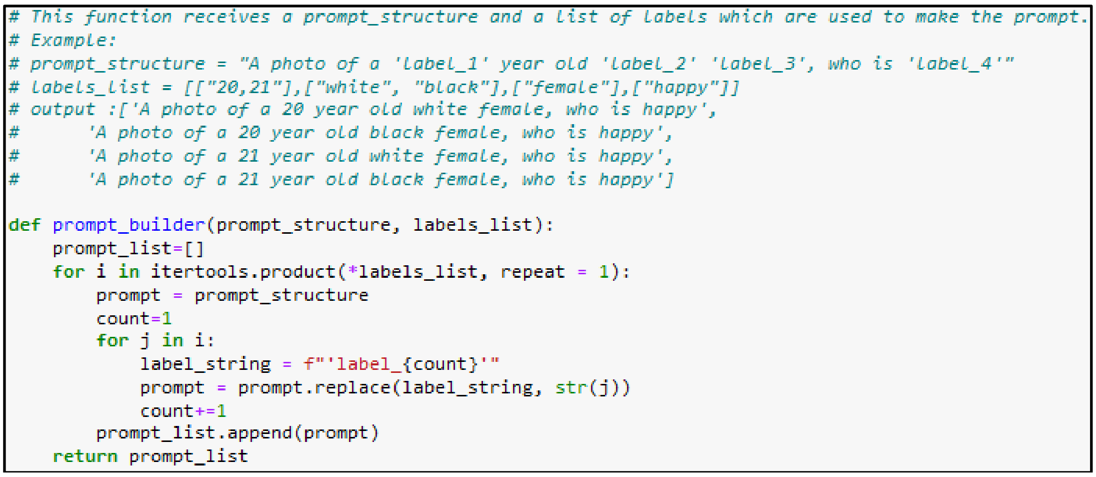
Chapter 6: Evaluation and Conclusion

Chapter one set the goal of :

> *"This project seeks to create a computer vision system that uses CLIP to associate labels to images of human faces, and which can become more accurate over time."*

Has this project achieved this goal, and has it done it well? First, I would argue that this project has achieved the goals set out in Chapter 1. A UI tool has been developed that can capture images from a user's webcam, detect multiple human faces in the image, and attach labels using CLIP-generated image embeddings. Moreover, although label method 3 is slow, it has shown positive improvement in accuracy as more data is added to the database. Thus, in that sense, the goals set out in chapter one have been achieved. However, it is important to acknowledge that this system is not ground-breaking and does not outperform anything already developed. The labelling process is slow and there are many potential efficiency improvements that could be made. For example, the labelling API returns labels for an image before being called again through the "/draw_labels" endpoint to draw a border around each face. This could be handled through one API call instead of two. I would also combine the Image Embedding API and the Labelling API into one API and simplify the logic greatly. This new, consolidated API would have a GeneralImageClassifier class. The GeneralImageClassifier would receive an image, a set of labels, and a prompt_structure, and would use the prompt_builder() function described below to build a dynamic list of prompts and feed them to the CLIP model. This approach would be much simpler, more efficient, and could automatically handle a broad range of image categories (you simply need to engineer the prompt correctly). If I had of taken this approach at the start of the project, I believe I would have achieved much more.

Therefore, future work could revolve around further experimentation with prompt engineering, which could improve the accuracy and range of labels predicted by CLIP. The prompt_builder() function shown in Figure 32 below could aid with this:



```
# This function receives a prompt_structure and a list of labels which are used to make the prompt.
# Example:
# prompt_structure = "A photo of a 'label_1' year old 'label_2' 'label_3', who is 'label_4'"
# labels_list = [["20,21"],["white", "black"],["female"],["happy"]]
# output :['A photo of a 20 year old white female, who is happy',
#       'A photo of a 20 year old black female, who is happy',
#       'A photo of a 21 year old white female, who is happy',
#       'A photo of a 21 year old black female, who is happy']

def prompt_builder(prompt_structure, labels_list):
    prompt_list=[]
    for i in itertools.product(*labels_list, repeat = 1):
        prompt = prompt_structure
        count=1
        for j in i:
            label_string = f"'label_{count}'"
            prompt = prompt.replace(label_string, str(j))
            count+=1
        prompt_list.append(prompt)
    return prompt_list
```

Figure 32: prompt_builder()

The prompt_builder() can be used in future work to generate a dynamic list of prompts for various image categories. For example, a list of prompts to label a dataset containing images of cars can be generated using the following prompt_structure and labels_list:

> **prompt_structure***: "*A photo of a *'label_1' 'label_2' 'label_3'*, priced between £*'label_4'* and £*'label_5'"*
> **labels_list***:* [["red", "blue", "black"], ["Toyota", "Ford", "BMW"], ["Avensis", "Focus", "X5"], ["10000", "20000", "30000"], ["40000", "50000", "60000"]].

This would produce a list of labels like :

> "A photo of a red Toyota Avensis, priced between £10000 and £40000".
> "A photo of a blue Toyota Avensis, priced between £10000 and £40000".
> "A photo of a black Toyota Avensis, priced between £10000 and £40000".
> Etc, etc….

If I were to start the project over again I would use this function as a basis for experimenting with prompt engineering and to develop the GeneralImageClassifier described above. Moreover, Although I spent some time developing a login system with user roles to make the UI tool a "product", I don't believe this is necessary as the UI tool was mainly created to support experimentation. Instead, I could have used that time on improving each component of the system and making sure they work well together.

Despite these limitations, I am content with the outcome of this project. While I have found it challenging, I have learnt a lot throughout the process, and it has contributed tremendously to my growth as a developer. Thus although there is much room for improvement, I am satisfied with the project overall, regardless of the final result.

References/Bibliography

Boesch, G. (2023) The 100 most popular computer vision applications in 2023, viso.ai. Available at: https://viso.ai/applications/computer-vision-applications/ (Last Accessed: April 11, 2023).

Deng, J. et al. (2019) Retinaface: Single-stage dense face localisation in the wild, arXiv.org. Available at: https://arxiv.org/abs/1905.00641 (Last Accessed: April 11, 2023).

Gallagher, N. and Neal, J. (2018) Necolas/normalize.css: A modern alternative to CSS resets, GitHub. Available at: https://github.com/necolas/normalize.css (Last Accessed: April 15, 2023).

Getting started with media devices (2020) WebRTC. Available at: https://webrtc.org/getting-started/media-devices (Last Accessed: April 15, 2023).

Howard, J. and Gugger, S. (2020) Deep learning for coders with FASTAI and pytorch, Deep Learning for Coders with Fastai and Pytorch: AI Applications Without a PhD. Google. Available at: https://books.google.no/books?id=xd6LxgEACAAJ (Last Accessed: April 11, 2023).

Karkkainen, Kimmo and Joo, Jungseock (2021) Fairface: Face attribute dataset for balanced race, gender, and age, FairFace: Face Attribute Dataset for Balanced Race, Gender, and Age for Bias Measurement and Mitigation. Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision. Available at: https://github.com/joojs/fairface (Accessed: April 13, 2023).

Klein, M. (2021) Flask vs. Django: Which framework should you choose?, Codecademy Blog. Available at: https://www.codecademy.com/resources/blog/flask-vs-django/ (Last Accessed: April 11, 2023).

Lihi Gur Arie, P.D. (2023) Clip: Creating image classifiers without data, Medium. Towards Data Science. Available at: https://towardsdatascience.com/clip-creating-image-classifiers-without-data-b21c72b741fa (Accessed: April 15, 2023).

IBM, MySQL vs. mongodb: What's the difference? (2022) IBM. Available at: https://www.ibm.com/cloud/blog/mysql-vs-mongodb (Accessed: April 15, 2023).

Liu, Z., Luo, P., Wang, X. and Tang, X. (2015) 'Deep Learning Face Attributes in the Wild', in Proceedings of International Conference on Computer Vision (ICCV), December 2015. Available at : http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html (Last Accessed 13/04/2023)

MDN Web Docs (2023) MediaDevices: GetUserMedia() method - web apis: MDN, Web APIs | MDN. Available at: https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia (Last Accessed: April 15, 2023).

MDN Web Docs (2023) Htmlcanvaselement: Todataurl() method - web apis: MDN, Web APIs | MDN. Available at: https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/toDataURL (Accessed: April 15, 2023).

Open AI (no date) Openai/CLIP: CLIP (Contrastive Language-image pretraining), predict the most relevant text snippet given an image, GitHub. Available at: https://github.com/openai/CLIP (Last Accessed: April 11, 2023).

Patil, V. (2019) Understanding faiss, Medium. Towards Data Science. Available at: https://towardsdatascience.com/understanding-faiss-619bb6db2d1a (Accessed: April 11, 2023).

Radford, A. et al. (2021) Learning transferable visual models from Natural Language Supervision, arXiv.org. Available at: https://arxiv.org/abs/2103.00020 (Last Accessed: April 11, 2023).

Serengil (2023) *Deepface/FaceDetector.py at master · Serengil/Deepface*, *GitHub*. Available at: https://github.com/serengil/deepface/blob/master/deepface/detectors/FaceDetector.py (Last Accessed: April 11, 2023).

Serengil et al. (2020) Deepface: A lightweight face recognition and facial attribute analysis (age, gender, emotion and race) library for Python, GitHub. Available at: https://github.com/serengil/DeepFace (Last Accessed: April 11, 2023).

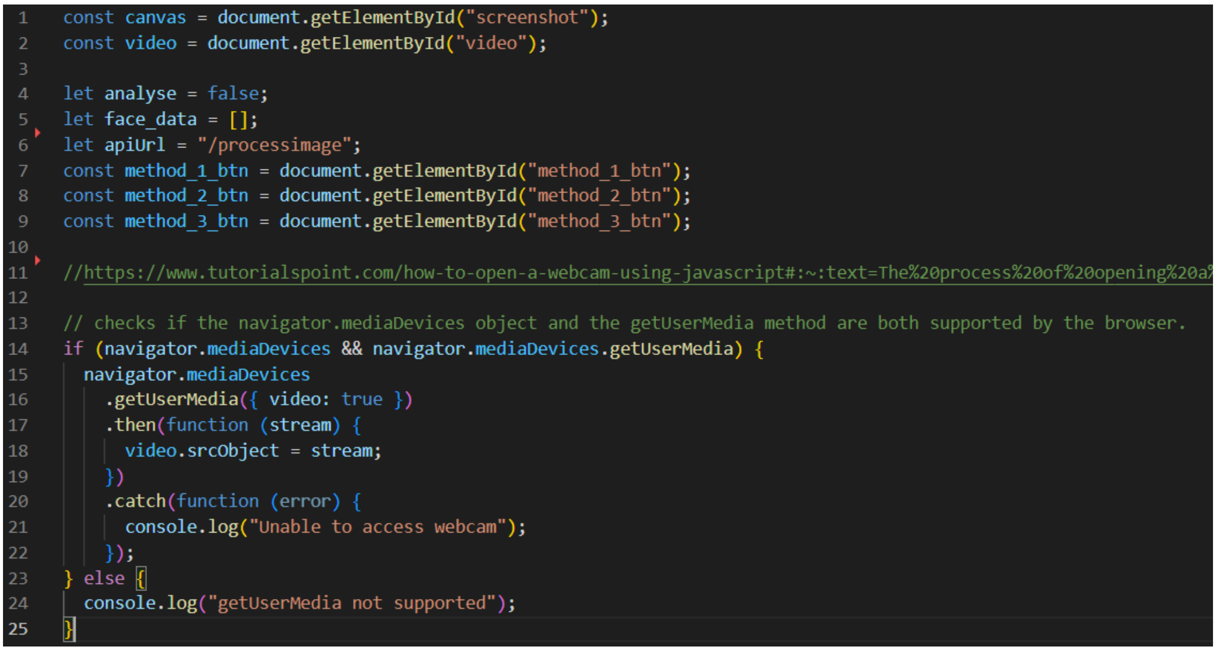
Welcome to FAISS documentation (2023) Welcome to Faiss Documentation - Faiss documentation. Available at: https://faiss.ai/ (Last Accessed: April 11, 2023).

What is Computer Vision? (no date) IBM. Available at: https://www.ibm.com/topics/computer-vision (Last Accessed: April 11, 2023).

Appendices

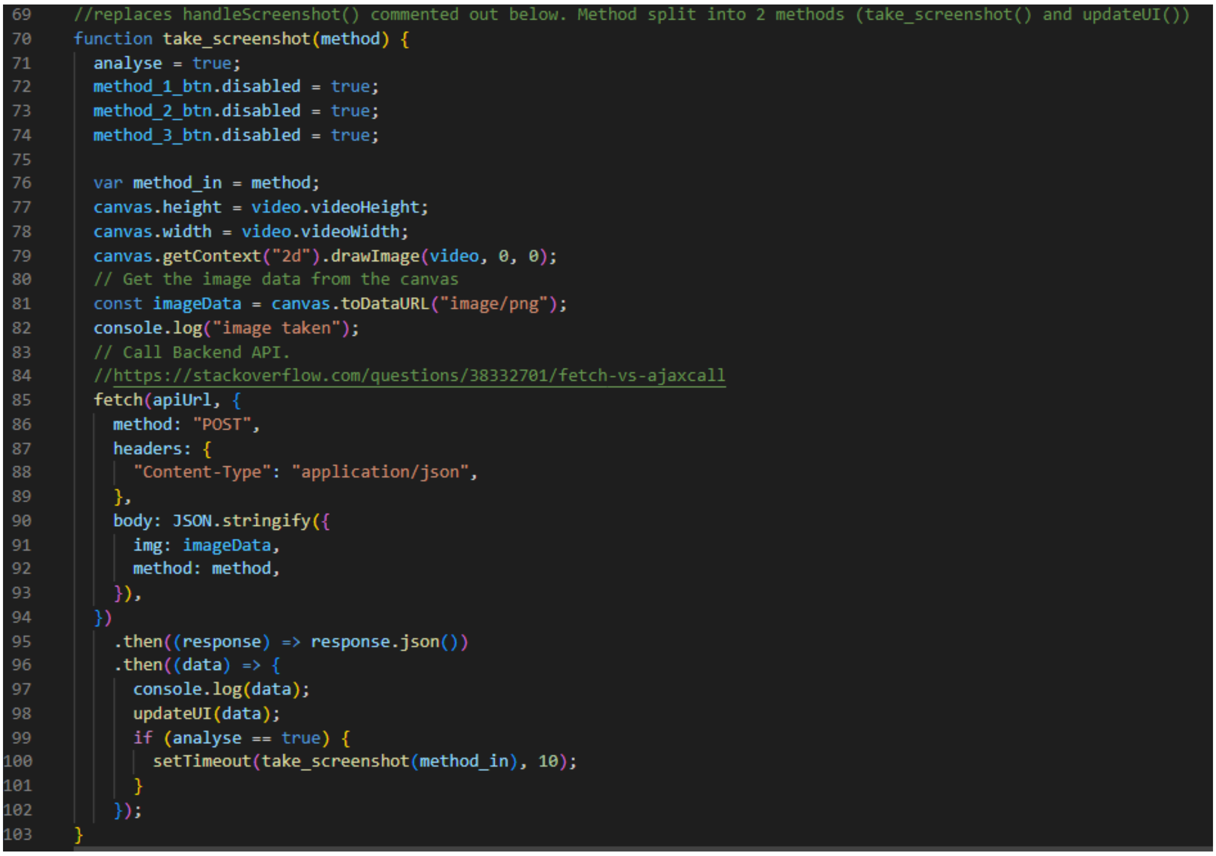
Code Snippets

Capture.js Figure 1:



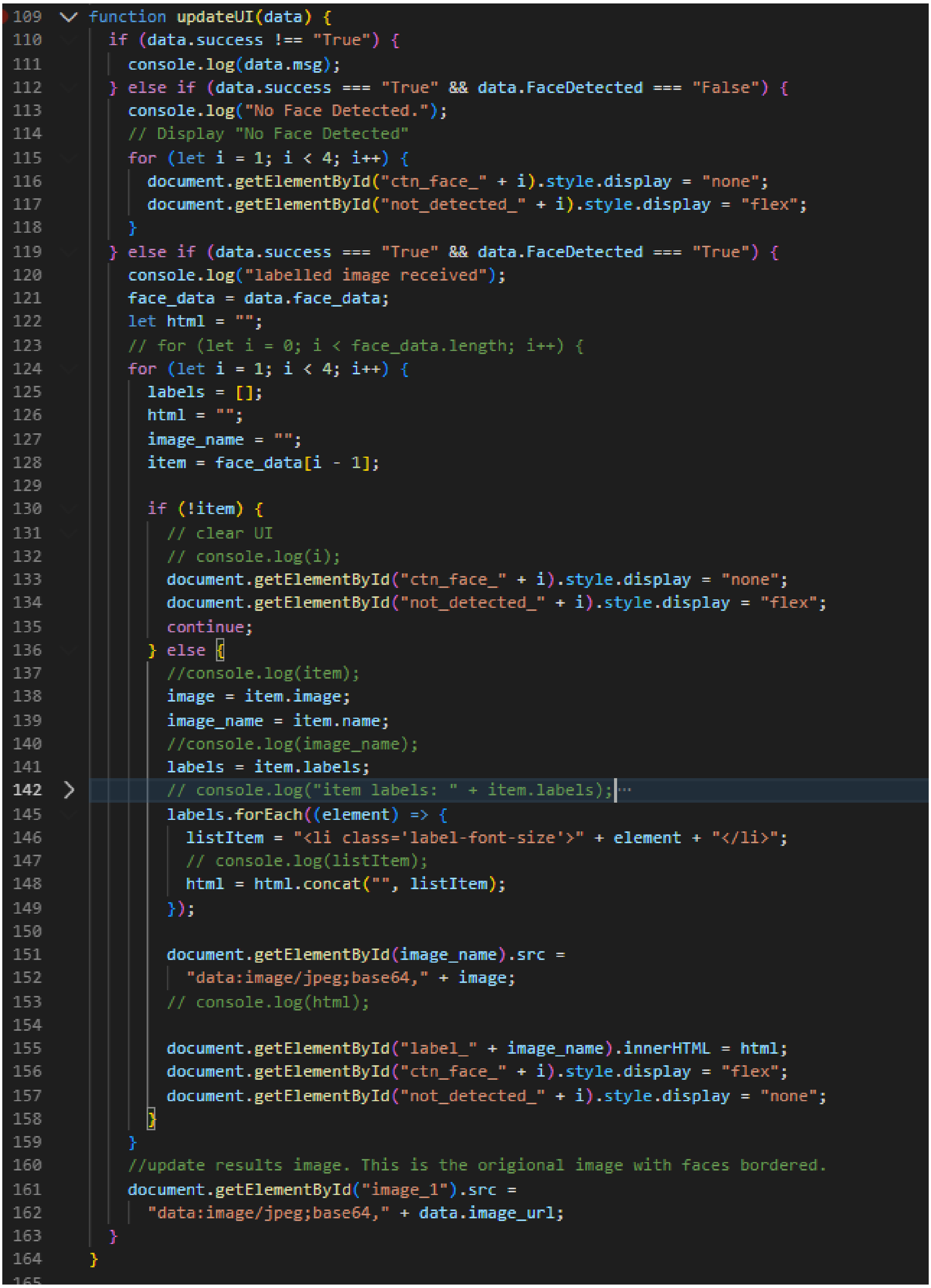
```
const canvas = document.getElementById("screenshot");
const video = document.getElementById("video");

let analyse = false;
let face_data = [];
let apiUrl = "/processimage";
const method_1_btn = document.getElementById("method_1_btn");
const method_2_btn = document.getElementById("method_2_btn");
const method_3_btn = document.getElementById("method_3_btn");

//https://www.tutorialspoint.com/how-to-open-a-webcam-using-javascript#:~:text=The%20process%20of%20opening%20a

// checks if the navigator.mediaDevices object and the getUserMedia method are both supported by the browser.
if (navigator.mediaDevices && navigator.mediaDevices.getUserMedia) {
  navigator.mediaDevices
    .getUserMedia({ video: true })
    .then(function (stream) {
      video.srcObject = stream;
    })
    .catch(function (error) {
      console.log("Unable to access webcam");
    });
} else {
  console.log("getUserMedia not supported");
}
```

Capture.js Figure 2:



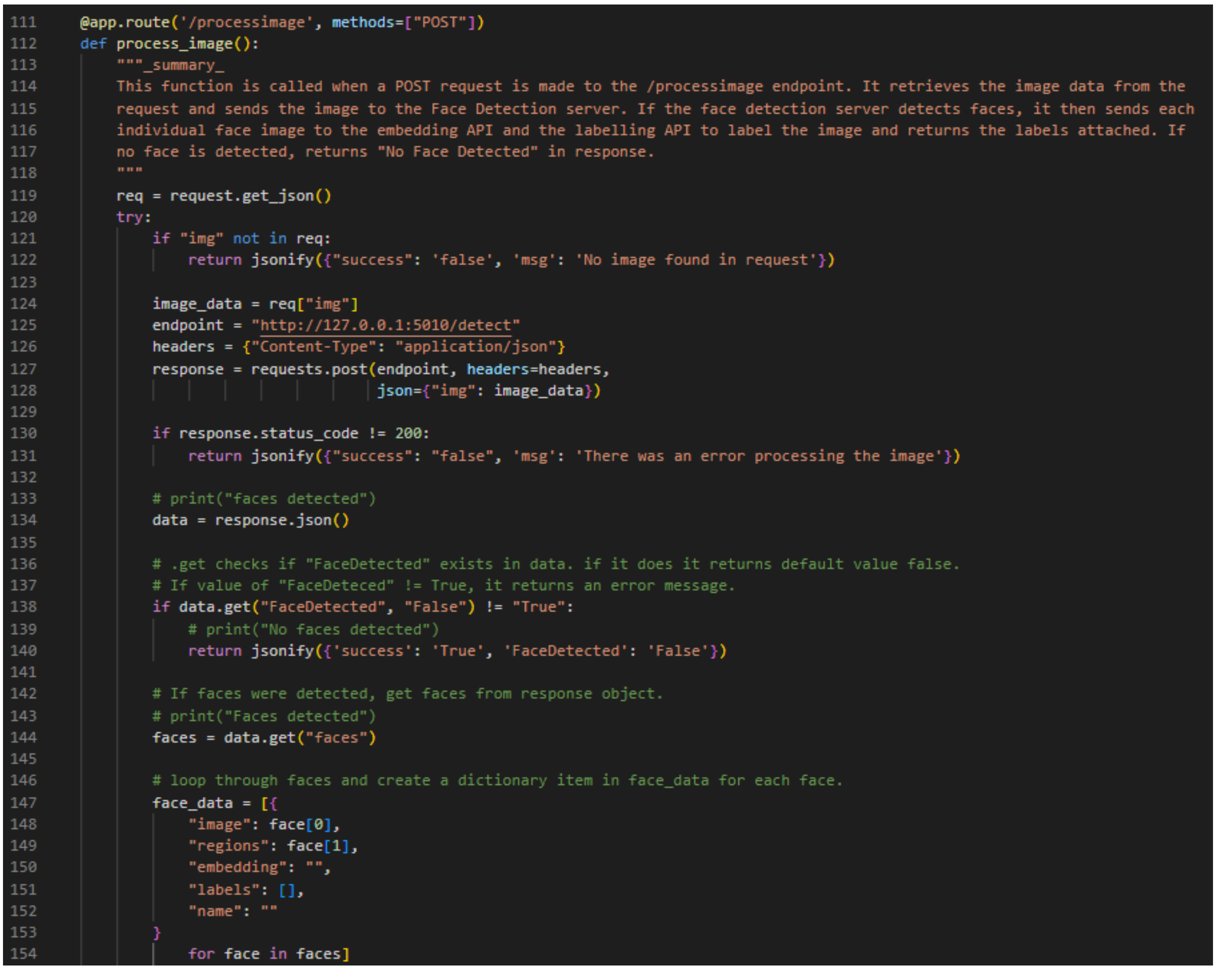
```
 //replaces handleScreenshot() commented out below. Method split into 2 methods (take_screenshot() and updateUI())
 function take_screenshot(method) {
   analyse = true;
   method_1_btn.disabled = true;
   method_2_btn.disabled = true;
   method_3_btn.disabled = true;

   var method_in = method;
   canvas.height = video.videoHeight;
   canvas.width = video.videoWidth;
   canvas.getContext("2d").drawImage(video, 0, 0);
   // Get the image data from the canvas
   const imageData = canvas.toDataURL("image/png");
   console.log("image taken");
   // Call Backend API.
   //https://stackoverflow.com/questions/38332701/fetch-vs-ajaxcall
   fetch(apiUrl, {
     method: "POST",
     headers: {
       "Content-Type": "application/json",
     },
     body: JSON.stringify({
       img: imageData,
       method: method,
     }),
   })
     .then((response) => response.json())
     .then((data) => {
       console.log(data);
       updateUI(data);
       if (analyse == true) {
         setTimeout(take_screenshot(method_in), 10);
       }
     });
 }
```

Capture.js Figure 3:



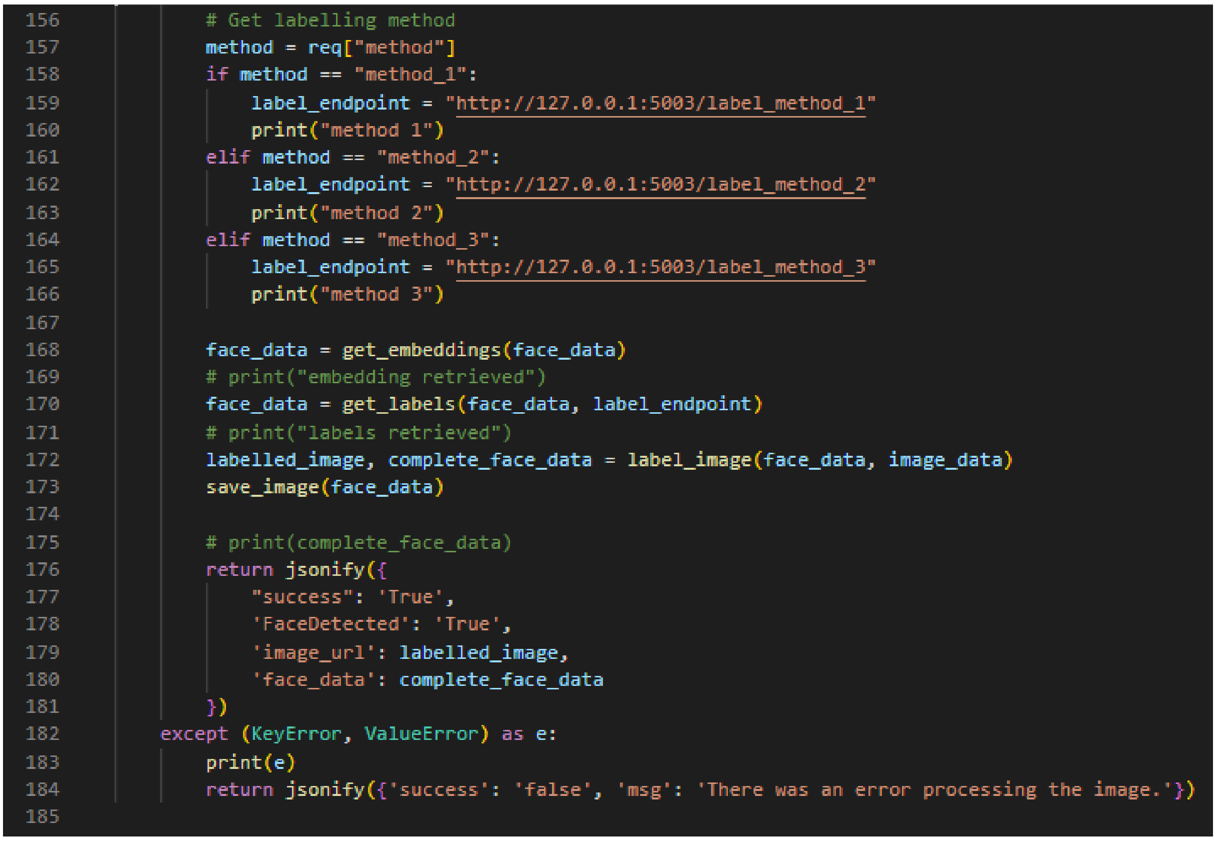
```
function updateUI(data) {
  if (data.success !== "True") {
    console.log(data.msg);
  } else if (data.success === "True" && data.FaceDetected === "False") {
    console.log("No Face Detected.");
    // Display "No Face Detected"
    for (let i = 1; i < 4; i++) {
      document.getElementById("ctn_face_" + i).style.display = "none";
      document.getElementById("not_detected_" + i).style.display = "flex";
    }
  } else if (data.success === "True" && data.FaceDetected === "True") {
    console.log("labelled image received");
    face_data = data.face_data;
    let html = "";
    // for (let i = 0; i < face_data.length; i++) {
    for (let i = 1; i < 4; i++) {
      labels = [];
      html = "";
      image_name = "";
      item = face_data[i - 1];

      if (!item) {
        // clear UI
        // console.log(i);
        document.getElementById("ctn_face_" + i).style.display = "none";
        document.getElementById("not_detected_" + i).style.display = "flex";
        continue;
      } else {
        //console.log(item);
        image = item.image;
        image_name = item.name;
        //console.log(image_name);
        labels = item.labels;
        // console.log("item labels: " + item.labels);
        labels.forEach((element) => {
          listItem = "<li class='label-font-size'>" + element + "</li>";
          // console.log(listItem);
          html = html.concat("", listItem);
        });

        document.getElementById(image_name).src =
          "data:image/jpeg;base64," + image;
        // console.log(html);

        document.getElementById("label_" + image_name).innerHTML = html;
        document.getElementById("ctn_face_" + i).style.display = "flex";
        document.getElementById("not_detected_" + i).style.display = "none";
      }
    }
    //update results image. This is the origional image with faces bordered.
    document.getElementById("image_1").src =
      "data:image/jpeg;base64," + data.image_url;
  }
}
```

‘/processimage’ route figure 1



```python
@app.route('/processimage', methods=["POST"])
def process_image():
    """_summary_
    This function is called when a POST request is made to the /processimage endpoint. It retrieves the image data from the
    request and sends the image to the Face Detection server. If the face detection server detects faces, it then sends each
    individual face image to the embedding API and the labelling API to label the image and returns the labels attached. If
    no face is detected, returns "No Face Detected" in response.
    """
    req = request.get_json()
    try:
        if "img" not in req:
            return jsonify({"success": 'false', 'msg': 'No image found in request'})

        image_data = req["img"]
        endpoint = "http://127.0.0.1:5010/detect"
        headers = {"Content-Type": "application/json"}
        response = requests.post(endpoint, headers=headers,
                                 json={"img": image_data})

        if response.status_code != 200:
            return jsonify({"success": "false", 'msg': 'There was an error processing the image'})

        # print("faces detected")
        data = response.json()

        # .get checks if "FaceDetected" exists in data. if it does it returns default value false.
        # If value of "FaceDeteced" != True, it returns an error message.
        if data.get("FaceDetected", "False") != "True":
            # print("No faces detected")
            return jsonify({'success': 'True', 'FaceDetected': 'False'})

        # If faces were detected, get faces from response object.
        # print("Faces detected")
        faces = data.get("faces")

        # loop through faces and create a dictionary item in face_data for each face.
        face_data = [{
            "image": face[0],
            "regions": face[1],
            "embedding": "",
            "labels": [],
            "name": ""
        }
            for face in faces]
```

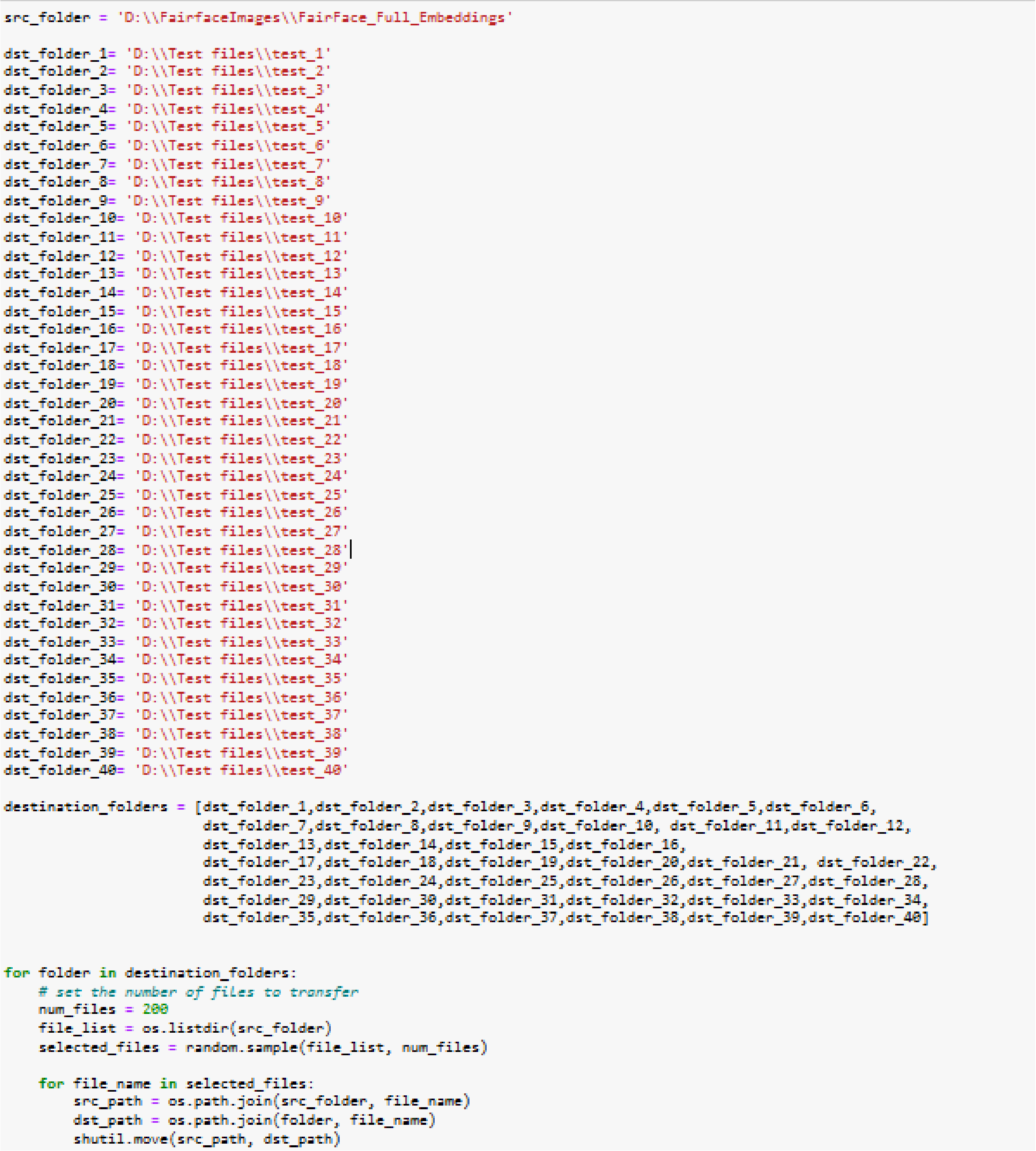
‘/processimage’ route figure 2



```python
        # Get labelling method
        method = req["method"]
        if method == "method_1":
            label_endpoint = "http://127.0.0.1:5003/label_method_1"
            print("method 1")
        elif method == "method_2":
            label_endpoint = "http://127.0.0.1:5003/label_method_2"
            print("method 2")
        elif method == "method_3":
            label_endpoint = "http://127.0.0.1:5003/label_method_3"
            print("method 3")

        face_data = get_embeddings(face_data)
        # print("embedding retrieved")
        face_data = get_labels(face_data, label_endpoint)
        # print("labels retrieved")
        labelled_image, complete_face_data = label_image(face_data, image_data)
        save_image(face_data)

        # print(complete_face_data)
        return jsonify({
            "success": 'True',
            'FaceDetected': 'True',
            'image_url': labelled_image,
            'face_data': complete_face_data
        })
    except (KeyError, ValueError) as e:
        print(e)
        return jsonify({'success': 'false', 'msg': 'There was an error processing the image.'})
```

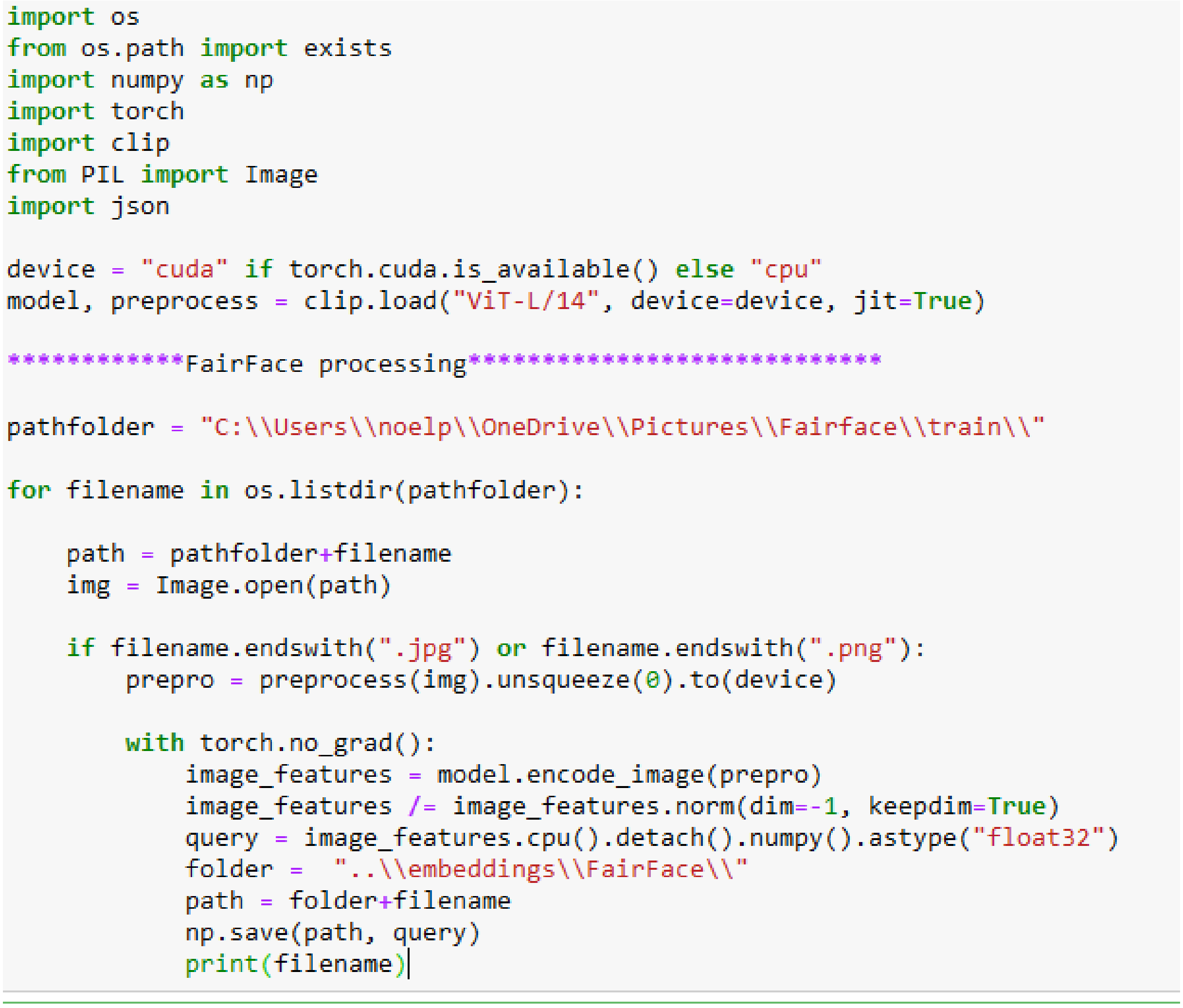
Scripts

Test File Setup Script:



```
src_folder = 'D:\\FairfaceImages\\FairFace_Full_Embeddings'

dst_folder_1= 'D:\\Test files\\test_1'
dst_folder_2= 'D:\\Test files\\test_2'
dst_folder_3= 'D:\\Test files\\test_3'
dst_folder_4= 'D:\\Test files\\test_4'
dst_folder_5= 'D:\\Test files\\test_5'
dst_folder_6= 'D:\\Test files\\test_6'
dst_folder_7= 'D:\\Test files\\test_7'
dst_folder_8= 'D:\\Test files\\test_8'
dst_folder_9= 'D:\\Test files\\test_9'
dst_folder_10= 'D:\\Test files\\test_10'
dst_folder_11= 'D:\\Test files\\test_11'
dst_folder_12= 'D:\\Test files\\test_12'
dst_folder_13= 'D:\\Test files\\test_13'
dst_folder_14= 'D:\\Test files\\test_14'
dst_folder_15= 'D:\\Test files\\test_15'
dst_folder_16= 'D:\\Test files\\test_16'
dst_folder_17= 'D:\\Test files\\test_17'
dst_folder_18= 'D:\\Test files\\test_18'
dst_folder_19= 'D:\\Test files\\test_19'
dst_folder_20= 'D:\\Test files\\test_20'
dst_folder_21= 'D:\\Test files\\test_21'
dst_folder_22= 'D:\\Test files\\test_22'
dst_folder_23= 'D:\\Test files\\test_23'
dst_folder_24= 'D:\\Test files\\test_24'
dst_folder_25= 'D:\\Test files\\test_25'
dst_folder_26= 'D:\\Test files\\test_26'
dst_folder_27= 'D:\\Test files\\test_27'
dst_folder_28= 'D:\\Test files\\test_28'
dst_folder_29= 'D:\\Test files\\test_29'
dst_folder_30= 'D:\\Test files\\test_30'
dst_folder_31= 'D:\\Test files\\test_31'
dst_folder_32= 'D:\\Test files\\test_32'
dst_folder_33= 'D:\\Test files\\test_33'
dst_folder_34= 'D:\\Test files\\test_34'
dst_folder_35= 'D:\\Test files\\test_35'
dst_folder_36= 'D:\\Test files\\test_36'
dst_folder_37= 'D:\\Test files\\test_37'
dst_folder_38= 'D:\\Test files\\test_38'
dst_folder_39= 'D:\\Test files\\test_39'
dst_folder_40= 'D:\\Test files\\test_40'

destination_folders = [dst_folder_1,dst_folder_2,dst_folder_3,dst_folder_4,dst_folder_5,dst_folder_6,
                       dst_folder_7,dst_folder_8,dst_folder_9,dst_folder_10, dst_folder_11,dst_folder_12,
                       dst_folder_13,dst_folder_14,dst_folder_15,dst_folder_16,
                       dst_folder_17,dst_folder_18,dst_folder_19,dst_folder_20,dst_folder_21, dst_folder_22,
                       dst_folder_23,dst_folder_24,dst_folder_25,dst_folder_26,dst_folder_27,dst_folder_28,
                       dst_folder_29,dst_folder_30,dst_folder_31,dst_folder_32,dst_folder_33,dst_folder_34,
                       dst_folder_35,dst_folder_36,dst_folder_37,dst_folder_38,dst_folder_39,dst_folder_40]


for folder in destination_folders:
    # set the number of files to transfer
    num_files = 200
    file_list = os.listdir(src_folder)
    selected_files = random.sample(file_list, num_files)

    for file_name in selected_files:
        src_path = os.path.join(src_folder, file_name)
        dst_path = os.path.join(folder, file_name)
        shutil.move(src_path, dst_path)
```

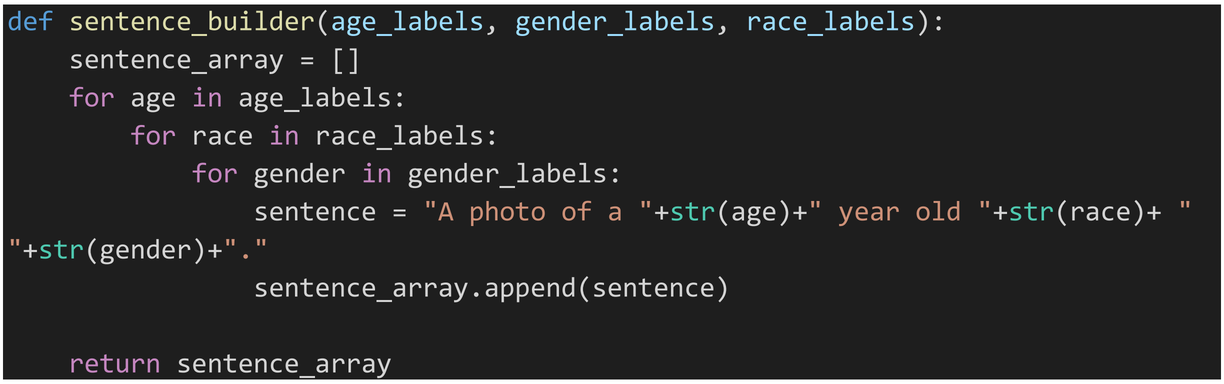
getEmbeddings.ipynb script:



```
import os
from os.path import exists
import numpy as np
import torch
import clip
from PIL import Image
import json

device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-L/14", device=device, jit=True)

************FairFace processing****************************

pathfolder = "C:\\Users\\noelp\\OneDrive\\Pictures\\Fairface\\train\\"

for filename in os.listdir(pathfolder):

    path = pathfolder+filename
    img = Image.open(path)

    if filename.endswith(".jpg") or filename.endswith(".png"):
        prepro = preprocess(img).unsqueeze(0).to(device)

        with torch.no_grad():
            image_features = model.encode_image(prepro)
            image_features /= image_features.norm(dim=-1, keepdim=True)
            query = image_features.cpu().detach().numpy().astype("float32")
            folder =  "..\\embeddings\\FairFace\\"
            path = folder+filename
            np.save(path, query)
            print(filename)
```

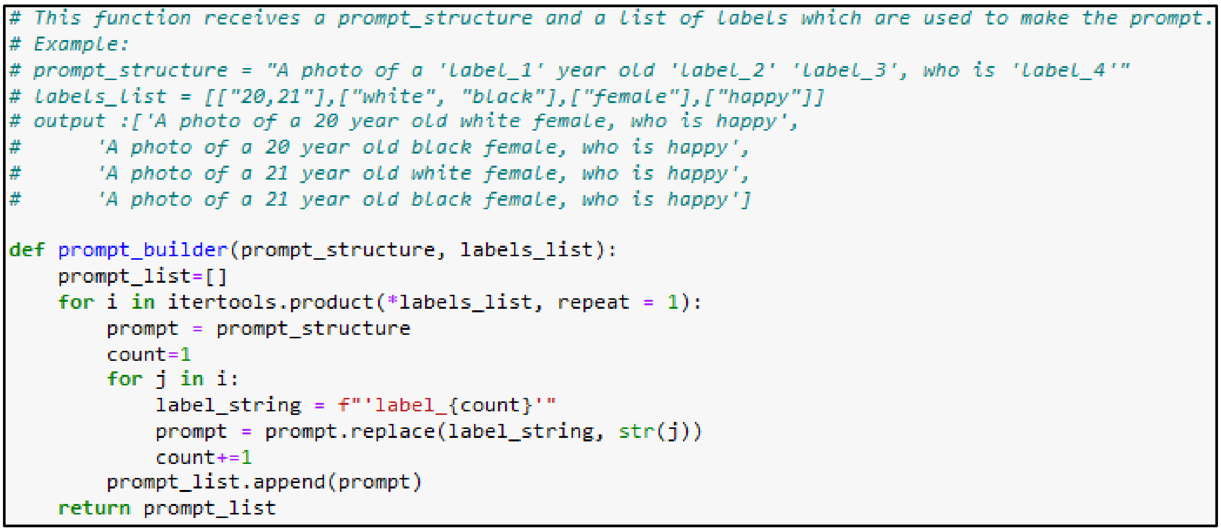
Functions

sentence_builder():



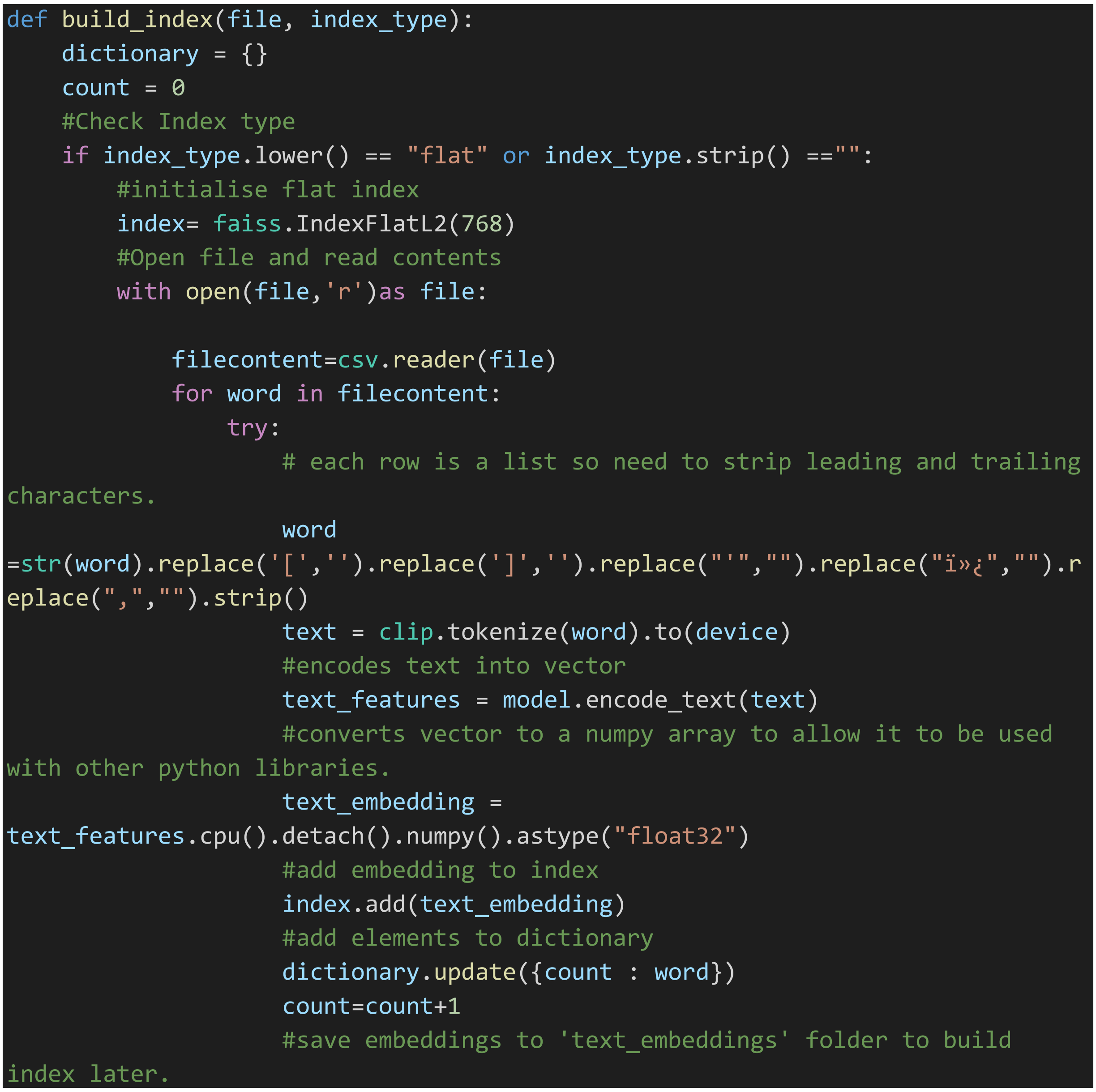
```
def sentence_builder(age_labels, gender_labels, race_labels):
    sentence_array = []
    for age in age_labels:
        for race in race_labels:
            for gender in gender_labels:
                sentence = "A photo of a "+str(age)+" year old "+str(race)+ "
"+str(gender)+"."
                sentence_array.append(sentence)

    return sentence_array
```

prompt_builder():



```
# This function receives a prompt_structure and a list of labels which are used to make the prompt.
# Example:
# prompt_structure = "A photo of a 'label_1' year old 'label_2' 'label_3', who is 'label_4'"
# labels_list = [["20,21"],["white", "black"],["female"],["happy"]]
# output :['A photo of a 20 year old white female, who is happy',
#         'A photo of a 20 year old black female, who is happy',
#         'A photo of a 21 year old white female, who is happy',
#         'A photo of a 21 year old black female, who is happy']

def prompt_builder(prompt_structure, labels_list):
    prompt_list=[]
    for i in itertools.product(*labels_list, repeat = 1):
        prompt = prompt_structure
        count=1
        for j in i:
            label_string = f"'label_{count}'"
            prompt = prompt.replace(label_string, str(j))
            count+=1
        prompt_list.append(prompt)
    return prompt_list
```

index_builder():

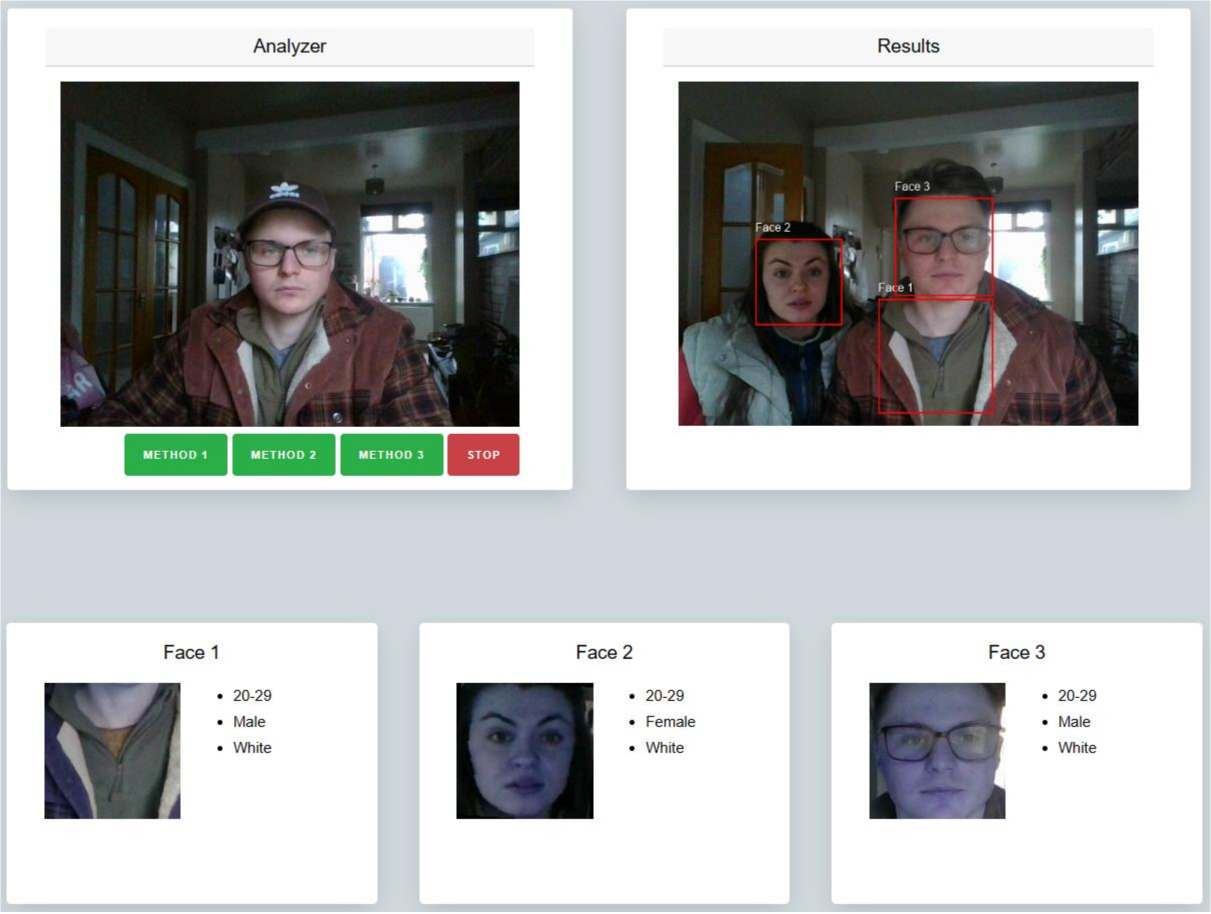


```
def build_index(file, index_type):
    dictionary = {}
    count = 0
    #Check Index type
    if index_type.lower() == "flat" or index_type.strip() =="":
        #initialise flat index
        index= faiss.IndexFlatL2(768)
        #Open file and read contents
        with open(file,'r')as file:

            filecontent=csv.reader(file)
            for word in filecontent:
                try:
                    # each row is a list so need to strip leading and trailing
characters.
                    word
=str(word).replace('[','').replace(']','').replace("'","").replace("ï»¿","").r
eplace(",","").strip()
                    text = clip.tokenize(word).to(device)
                    #encodes text into vector
                    text_features = model.encode_text(text)
                    #converts vector to a numpy array to allow it to be used
with other python libraries.
                    text_embedding =
text_features.cpu().detach().numpy().astype("float32")
                    #add embedding to index
                    index.add(text_embedding)
                    #add elements to dictionary
                    dictionary.update({count : word})
                    count=count+1
                    #save embeddings to 'text_embeddings' folder to build
index later.
```

```
                    # path =
folder+str(word).replace('[','').replace(']','').replace("'","")
                    # np.save(path, text_embedding)
                except Exception as e:
                    print(e)
                    return "There was an error."
    else:
        #build logic for another type of index
        return ""
    return index, dictionary
                    # path =
folder+str(word).replace('[','').replace(']','').replace("'","")
                    # np.save(path, text_embedding)

                except Exception as e:
                    print(e)
                    return "There was an error."
    else:
        #build logic for another type of index
        return ""


    return index, dictionary
```

Known Issues

1. There is a known issue where the face detector sometimes detects phantom faces. This is worse in low light conditions. Occurrences of this error can be reduced by changing the detector back end. However, this also has the effect of increasing latency.



Face 1

- 20-29
- Male
- White

Face 2

- 20-29
- Female
- White

Face 3

- 20-29
- Male
- White