Application for live facial recognition

Mia Brown

Harrisburg University of Science and Technology

CISC 498: Project II

Fall 2024

Abstract

This project explores the development of a facial recognition system integrated with an Android application. The application enables users to upload images for facial recognition through a live camera feed and sends the data to a Flask server for processing. The system employs a trained k-Nearest Neighbors (k-NN) classifier for recognition, leveraging a dataset of images. The Android app uses Retrofit for seamless communication with the Flask server and incorporates key features like live feed, camera interaction, and user-friendly interface design. The paper discusses the technical aspects of building the app, the challenges faced during implementation, and the ethical considerations associated with facial recognition technology. While the project achieved partial success, it fell short of the original vision due to limitations with camera integration and database use. This paper concludes with reflections on the project's outcomes and insights into potential future improvements.

I. Introduction

Facial recognition technology has rapidly advanced in recent years, with applications spanning from security systems to personal devices. For this project, I set out to design an Android-based facial recognition system that could be easily used for personal security purposes. The primary objective was to develop an app capable of identifying individuals from live camera feeds and images, making it both functional and user-friendly. The project that has been developed consists of an application that has the capability to perform facial recognition live from a captured image. The development process required combining several technical components, including an Android application, a Flask-based backend server, and a machine learning model for facial recognition. These elements work together to provide users with the ability to capture or upload images for recognition. The Flask server processes the images using a pre-trained k-NN classifier, returning results in real time. Despite initial setbacks, including challenges with camera integration and database implementation, the project successfully achieved a simplified version of its intended functionality. Alongside technical development, this paper explores the ethical and privacy implications of using facial recognition technology, particularly the potential for misuse and data security concerns. This work represents not only the creation of a functional application but also a learning experience in software design, machine learning implementation, and navigating the complexities of ethical technology use. Looking forward, there is significant potential to expand and refine the project, incorporating features like database integration and improved camera compatibility to achieve the original vision.

II. Methodology

This section of the paper will cover the details that go along with the development of the application. There are many different avenues that were ventured down, and this will explore each one and where each one faltered. The backbone of this project is the application that was created for the end users' use. I wanted to be able to create an app that can use uploaded images from the app and use facial recognition live in a camera view that is also on the application.

2.1. UI Application Development

This being the backbone of the project, this application needed to be user friendly and successfully connect to a flask server to then send the images for comparison against a trained model of images. In figure 1. you will see the home page of the Application**.** I aimed to create a straightforward and user-friendly application that would be simple to navigate.

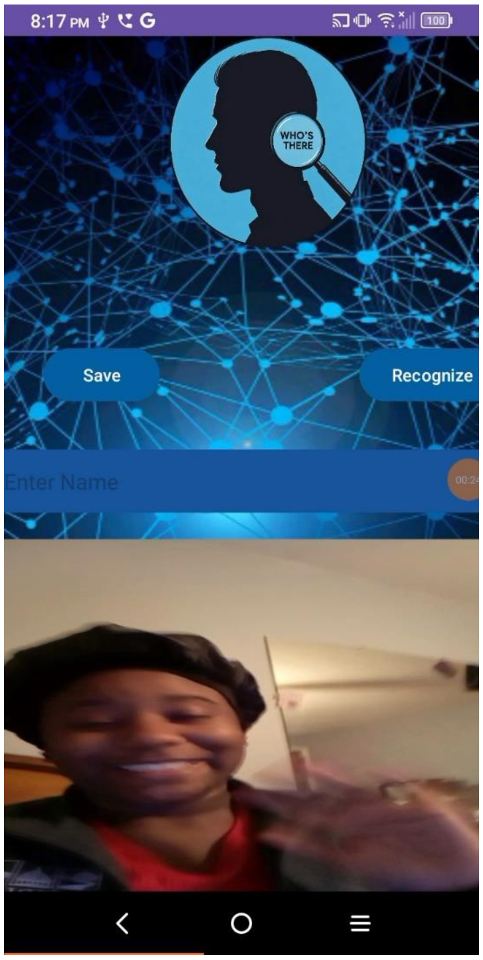


Figure 1: Home Screen of app

At the bottom of the home screen, a live video feed is displayed, sourced from an external Android phone emulating the application. The user has the option to initiate face recognition by selecting the "Recognize" button. Once chosen, the application activates the camera, allowing the user to select either the front or rear camera to capture an image. After the image is captured, as shown in Figure 2, a checkmark appears to indicate that the photo is ready to be sent for recognition. Upon completion of the recognition process, a toast message will appear, notifying the user of the identified person or indicating if the face is unknown, as seen in Figure 3. I tested the application after adding myself to the database of known faces, and as demonstrated in Figure 4, the system successfully recognized me, Mia, as the person in the image.

Going deeper, let's dive into what makes up the main activity of the app. Main Activity is the entry point of the application. It also manages the UI, camera interactions, and server communication. First it has check permissions to access the camera. If the app does not have the permissions, it asks the user to allow camera access, which is called using the *Manifest.permission.CAMERA*. A list of available cameras is retrived from a call of *StartCamera()*, and a check for a front-facing camera, *(Lens_Facing_Front)*, by using the

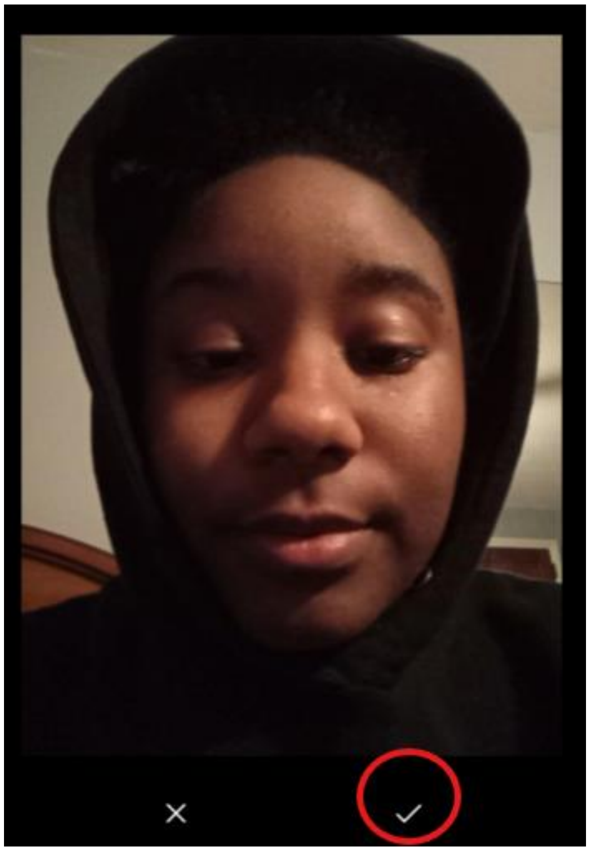
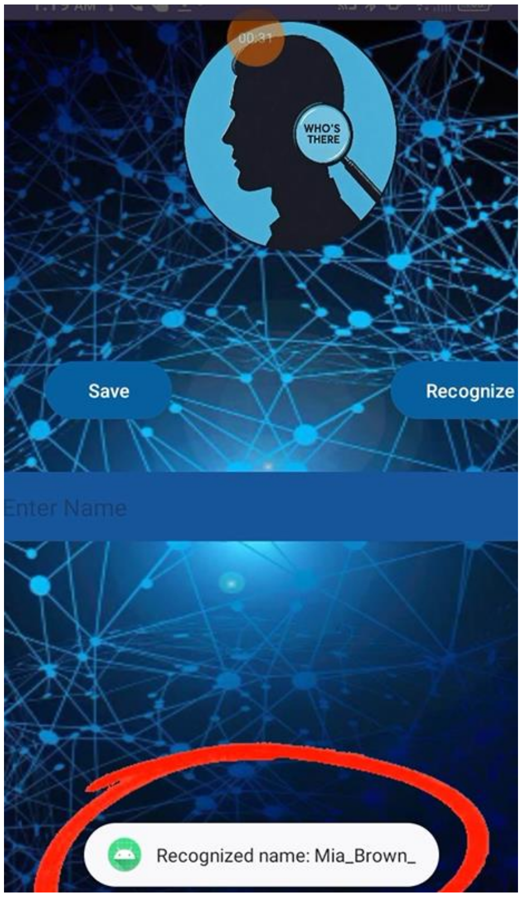


Figure 2: Sending image for recognition

openCamera method. The *StartCamera* button launches a camera intent *(MediaStore.ACTION_IMAGE_CAPTURE)* to capture an image. The captured image is returned as a Bitmap, which is saved locally in a designated directory using *FileOutputStream.* The file path is stored as a URI for further use. The app also includes a helper method to retrieve the real file path from the URI, ensuring compatibility with different Android devices. After saving the image, the app uses Retrofit to communicate with a Flask server for facial recognition. The image is converted into a multipart file using *RequestBody* and sent via a POST request. The server processes the image and returns a response containing the recognized name, which is displayed to the user via a *Toast* message.

The next part of this app that is extremely important is the flask api. This establishes a seamless communication between the android app and the flask server which will handle facial recognition. The *FlaskApi* interface includes a method, *recognizeFace*, annotated with Retrofit's @Multipart and @POST directives. These annotations specify that the request is a multipart HTTP POST targeting the /recognize endpoint on the Flask server. This configuration is essential for transferring image files efficiently and reliably. The method accepts a *MultipartBody.Part* parameter, which encapsulates the image file. Retrofit ensures the image file is appropriately formatted and sent as part of the request payload. The server processes the incoming image, performs facial recognition, and responds with a structured *RecognitionResponse*, which contains the results of the recognition process.

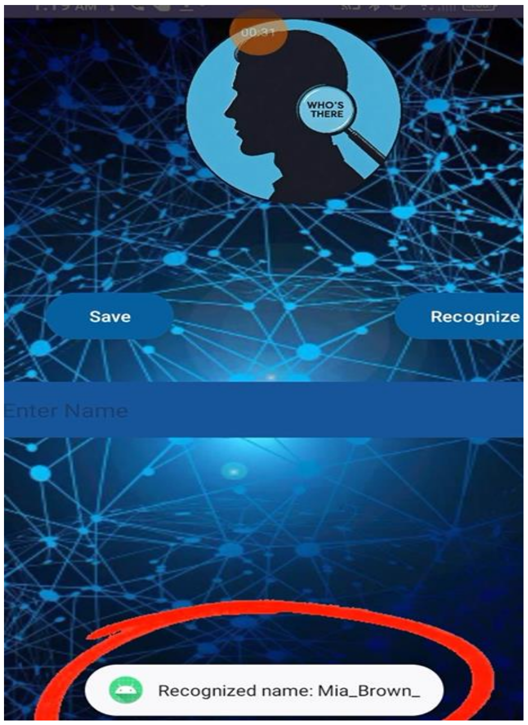


Recognized name: Mia_Brown_

Figure 3: Name Recongized - Mia

A big part of the application relies on the Retrofit client, which handles all the communication between the Android app and the Flask server. Without it, the app wouldn't be able to send images to the server or receive any facial recognition results. The Retrofit client simplifies making HTTP requests, letting us focus on the app's features instead of worrying about the technical details of network communication. The *RetrofitClient* class starts by making sure there's only one instance of Retrofit running, which is smart because it saves resources and keeps everything consistent across the app. This is done with a simple check: if there's no

Retrofit instance yet, it creates one; if it’s already there, it just uses that. This setup is known as the singleton pattern. It’s a reliable way to ensure all network calls go through the same configured instance. Inside the class, there’s an OkHttpClient that’s used for handling all the HTTP operations. It’s customized to include a logging interceptor, which is super helpful for development. This interceptor logs every HTTP request and response, including their bodies. It’s like having a behind-the-scenes look at what the app is sending to and getting back from the server. This makes debugging a lot easier because you can quickly see if something’s wrong with the data being sent or received. The *Retrofit.Builder* is what pulls everything together. It sets the server’s base URL, which in this case is http://10.0.0.202:5000/. This base URL is where all the API endpoints live, so every request the app makes is automatically directed there. The builder also includes a *GsonConverterFactory*, which handles converting JSON data from the server into Java objects the app can use, and vice versa. This means when the Flask server sends a JSON response with recognition results, the app doesn’t have to manually parse it, it’s already in a usable format. In short, the Retrofit client is what makes the app and server "talk" to each other smoothly. It’s set up to be efficient, reusable, and easy to debug, which is important when you’re dealing with real-time processes like facial recognition. By taking care of all the heavy lifting for network communication, the Retrofit client ensures the app can perform its tasks without hiccups. It’s a small piece of code that plays a big role in keeping everything connected and running smoothly.



Recognized name: Mia_Brown_

Figure 4: Name Recognized - Mia

2.2. Facial Recognition Implementation

This stage of the project was the most challenging and caused uncertainty for me. I was unsure about how the facial recognition component would turn out, given the complexity of the task. I initially attempted to create my own facial recognition code for training, but after extensive research, I found little success. Eventually, I discovered a Python-based solution for facial recognition that aligned with the research I had done in the earlier stages of my project. Specifically, I sought out code that incorporated *matplotlib* and *cv2* to process the images. The chosen code was ideal for my needs, as it extracted images from a ZIP file, processed them to create a dataset, detected and encoded faces, and then trained a k-Nearest Neighbors (KNN) classifier. Once the training was complete, it saved the model and labels as separate pickle files, which I would later pass to the Flask server for further processing. A key part of the implementation involved adding my own images to the dataset. To ensure consistency and accuracy in the classification and recognition process, I first examined the existing dataset closely. Each image was 160x160 pixels and cropped tightly around the face, with each person having approximately 75 images. The file names followed a specific format: *firstname_lastname_0.jpg*, with the number increasing sequentially as more images were added. I made sure to take my own images according to these exact specifications to maintain the integrity of the dataset and ensure that the training and recognition processes were as precise as possible.

2.3. Flask Server

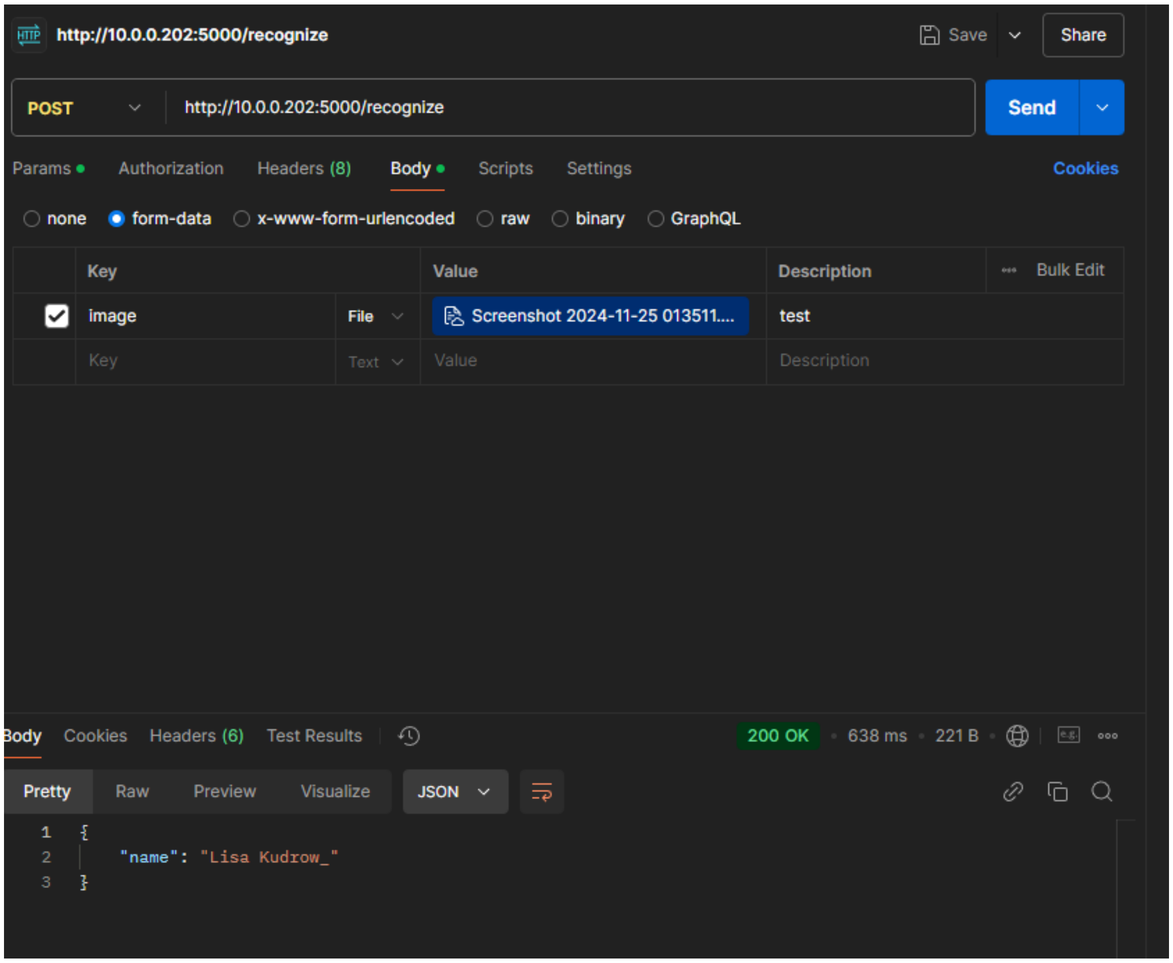
This Flask-based application serves as a backend server for a facial recognition system, leveraging a pre-trained k-Nearest Neighbors (k-NN) classifier to identify faces. It is designed to handle HTTP POST requests sent to the /recognize endpoint, with images uploaded via the image field in the request. The server begins by validating the presence of the image file in the request and then processes it using the face_recognition library to detect facial landmarks and encode facial features into a numerical vector representation. The application enforces a constraint that exactly one face must be present in the image, which is critical to avoid ambiguity or misclassification. If the face detection fails or multiple faces are found, the server responds with an appropriate error message. Once a face is successfully encoded, the numerical vector is passed as input to the k-NN model, previously trained and serialized into a knn_model.pkl file. The model predicts the most likely class label (the individual’s name) and provides a probability distribution for the prediction, indicating the confidence levels for all possible labels. To ensure reliable recognition, the system evaluates the highest confidence value against a predefined threshold (0.6). If the confidence is below this threshold, the system classifies the face as "unknown," prioritizing accuracy over false positives. If the threshold is met or exceeded, the server responds with the predicted name of the individual, encapsulated in JSON format. This predicted name is taken from the striped file names that are stored in the labels.pkl file.

2.4. KNN Classifier

In my project, I decided to modify the original code by implementing the KNN classifier to improve the accuracy and efficiency of the machine learning model. I used a KNN classifier from Kaggle as the base, making several changes to fit the needs of my specific application. The original code was using a simpler model that didn’t incorporate the sophisticated features of the KNN algorithm. My changes involved introducing the KNN classifier and adjusting the model training process. By doing so, I was able to take advantage of KNN's strength in handling classification problems by analyzing the closest neighbors of a data point, which allows for better performance in predicting classes. Additionally, I worked on pre-processing the input data. I ensured that the dataset was cleaned properly, including dealing with missing values and scaling the features to ensure they all had equal weight in the KNN classification process. The original code didn’t emphasize these aspects, which I considered necessary to improve the model’s accuracy. The process began by extracting images from a ZIP file containing face images, which were then organized into a directory structure. The filenames of the images were used to extract labels, with the assumption that the filename contained the person's name. I employed Python’s zipfile module to handle the extraction, and by using the os library, I navigated the directory to list all image paths for further processing. Once the images were extracted and labeled, I proceeded with the face detection and encoding phase. For each image, I used the face_recognition library, which allows for robust face detection. Initially, the face detection was performed using the HOG (Histogram of Oriented Gradients) model for efficiency, and if no faces were detected, the system fell back to using the CNN-based model for better accuracy. Images with multiple faces or no faces were discarded to ensure the dataset contained only clear, single-face images, which were then processed to obtain face encodings. With the face encodings extracted from the images, I created the dataset by pairing the encodings with the corresponding labels. I then split the dataset into training and testing sets using scikit-learn's train_test_split() function. This split ensured that the classifier could be trained on one portion of the data and evaluated on another. I chose to use the KNN classifier due to its simplicity and effectiveness in tasks like face recognition. The KNN model was trained with the training set and evaluated using standard metrics such as precision, recall, and F1-score, which helped assess its classification performance. I also visualized the results using a confusion matrix, providing further insight into how well the model was able to classify faces correctly. Once the model was trained, I tested its ability to recognize faces in new, unseen images. After detecting faces in these images, I applied the trained KNN model to predict the identities of the faces. A threshold distance of 0.6 was used to determine if a match was found. If a match was detected, the person's name was predicted; otherwise, the system labeled the face as "unknown." The system also annotated the images by drawing rectangles around the detected faces and labeling them with the predicted names, which were displayed for verification.

Results

The results of the facial recognition tests exceeded my expectations. Before testing with the actual application code, I first conducted preliminary tests using Postman to validate the recognition functionality. Figure 4 displays the outcome of the recognition test.



POST http://10.0.0.202:5000/recognize

Body: [✓ form-data]

	Key		Value	Description
✓	image	File	Screenshot 2024-11-25 013511....	test
	Key	Text	Value	Description

200 OK · 638 ms · 221 B

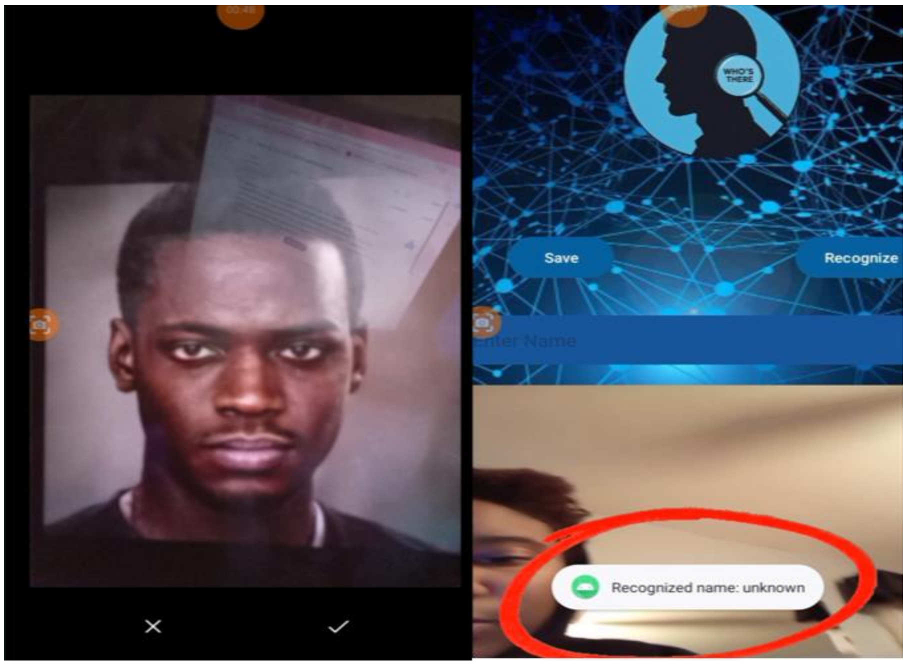
JSON

```
{
    "name": "Lisa Kudrow_"
}
```

Figure 5: Postman Face Recognition Test

For the initial test, I used an image of Lisa Kudrow. In Postman, the HTTP request was set to POST, and the URL along with the POST name "recognized" was specified. In the body of the request, I included a key labeled "image" with the value being a downloaded image of Lisa Kudrow, which was not part of the dataset. Upon submitting the request, the system successfully returned the guessed name, which was extracted from the name labels stored in the pickle file. The results, as shown at the bottom of Figure 4, demonstrated that the recognition was able to accurately identify the image.

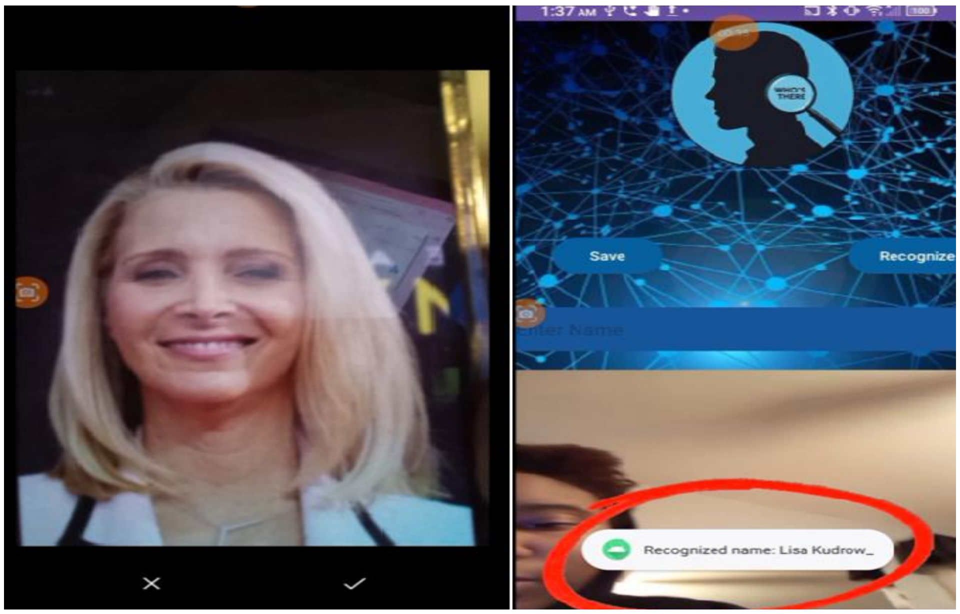
Following the successful Postman test, I proceeded to test the application. I used a mix of faces from the dataset, faces not present in the dataset, and my own image. The first test involved an "unknown" face, which is shown in Figure 5. On the left of the image, the captured person is displayed, while on the right, the expected toast message indicates the result: "Unknown." This outcome was expected, as the face in question was not in the dataset, leading to the classification as "unknown."



Recognized name: unknown

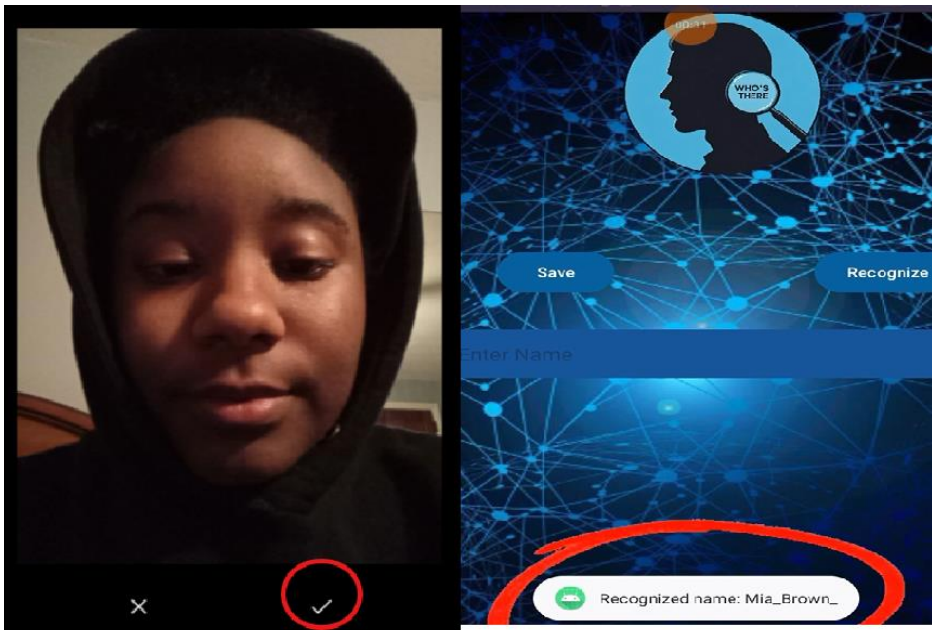
Figure 6: Unknown Face Rec with App

Subsequent tests involved images from the dataset. Figure 6 demonstrates the results with an image of Lisa Kudrow, which was also not part of the original dataset. The recognition returned the correct identification, confirming that the system was functioning as expected. Finally, I tested with my own image, and the results were equally successful, as seen in Figure 7. These tests validate that the facial recognition system performs as intended, accurately identifying faces from the dataset while correctly classifying unknown faces.



Recognized name: Lisa Kudrow_

Figure 7: Face Test with Known face In App



Recognized name: Mia_Brown_

Figure 8: Test my face "known" in App

Project fallbacks

The original vision for this project underwent significant changes as I encountered various challenges during development. My initial plan was to create an application that connected to an external motion detection camera, with the goal of displaying the live feed within the app. Additionally, I intended to allow users to add faces to a database for recognition purposes. Once the database was populated, the app was designed to automatically detect and display the name of the person present in the live feed. However, several issues prevented this plan from coming to fruition. The first major obstacle was connecting the camera to the application. Wyze cameras, which I had purchased, require the use of RTSP (Real-Time Streaming Protocol) to connect to third-party applications. Unfortunately, the camera I selected did not support RTSP, making it impossible to establish the necessary connection. As a result, I had to pivot and opt for facial recognition through the phone's camera. Another challenge arose with the data collection process. The MySQL database setup for storing images was not sufficient for running facial recognition effectively, as the database contained only around 50 images, leading to issues and bugs. To resolve this, I chose to use a pre-existing dataset from Kaggle, which provided a much larger and more suitable collection of images for the recognition process.

Discussion and Ethics

When it comes to facial recognition, legal and ethical issues are significant concerns that are becoming increasingly prevalent in society. Since facial recognition technology can be used in public spaces, it can potentially violate privacy rights by collecting and processing biometric data without an individual's consent or knowledge. This data can be gathered through apps, security cameras, or other surveillance tools, often in public areas, which brings up critical questions regarding the extent to which personal information is collected, how it's stored, and for what purpose. There are ongoing debates about where this data is stored, who has access to it, and how it might be used in the future. For instance, when facial recognition technology is used by law enforcement or government agencies for surveillance, it often leads to concerns about civil liberties being infringed upon and the potential for abuse of power. In some cases, individuals might be unaware that their faces are being scanned and stored, potentially leading to surveillance that violates their right to privacy. For my project, these ethical issues are particularly relevant. In the current design of the application, any person’s face can be added to the dataset for facial recognition, provided there is an image available. This means that without proper safeguards in place, anyone’s image could be uploaded, duplicated, and added to the database, potentially without their explicit consent. As the app evolves, it will be important to address these concerns, ensuring that the users are fully informed about the use of their biometric data and that the application complies with privacy regulations and ethical guidelines.

Closing statements and Future of project

After finishing this project, I was a bit disappointed that the outcome didn’t match my original vision. While the app has basic functionality, some key features didn’t come together as I had hoped. However, I’m determined to implement my original idea and continue working on it. One of the main changes I want to make is using a MySQL database to store facial recognition images instead of storing them locally on the device. This would streamline the process and help the app scale better. Another improvement I want to add is the ability for users to add new faces to the database. Currently, if the system detects an "unknown" face, the user can’t take any action. I plan to let users enter the name of the person and save it directly from the home screen using an edit text field. Once the name is entered and the "save" button is pressed, the app would upload the image to the database as multiple copies (about 75 images) to improve recognition accuracy. To make this work, I’ll use "watchdog," a tool that monitors the app for changes and triggers actions when there’s an update. When the new image is uploaded, the watchdog will activate the k-Nearest Neighbors (KNN) classifier to re-train and include the new face in the system, allowing the app to continuously improve. These changes would make the app more interactive, allowing users to update the database and making the system more personalized. I’m excited to continue developing the app and bring my original vision to life.

Work Cited

Python, R. (n.d.). *Face Detection in Python Using a Webcam – Real Python*. Realpython.com. https://realpython.com/face-detection-in-python-using-a-webcam/

iamscottxu. (2023, December 25). *GitHub - iamscottxu/obs-rtspserver: RTSP server plugin for obs-studio*.
GitHub. https://github.com/iamscottxu/obs-rtspserver

Labs, W. (2024, October 9). *RTSP firmware for Wyze Cam V4*. Wyze Forum. https://forums.wyze.com/t/rtsp-firmware-for-wyze-cam-v4/312457/6

RTSP. (2024). Android Developers.

https://developer.android.com/media/media3/exoplayer/rtsp#java

Maven Repository: com.google.android.exoplayer» exoplayer-rtsp» 2.19.1. (2023).

Mvnrepository.com.

https://mvnrepository.com/artifact/com.google.android.exoplayer/exoplayer-rtsp/2.19.1

Mahbub. (2023, April 30). *RTSP: The Real-Time Streaming Protocol Explained*. Castr's Blog.
https://castr.com/blog/what-is-rtsp/

Face Recognition Dataset. (n.d.). Www.kaggle.com.
https://www.kaggle.com/datasets/vasukipatel/face-recognition-dataset?resource=downloadhttps://www.kaggle.com/code/andrawesashraffarouk/face-recognation/notebook

Klejvi Kapaj. (2023, July 3). *Changing Retrofit Base URL at Runtime Using a Singleton Class*.

Medium.

https://medium.com/@klejvisiper/changing-retrofit-base-url-at-runtime-using-a-singleton-class-86013b7e3c30

Simple. (2022, January 6). *Creating a Simple Python Web Application With Flask and Testing Locally*. YouTube. https://youtu.be/Aa-F6zqLmig

GeeksforGeeks. (2021, January 13). Cropping Faces from Images using OpenCV Python.
GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/cropping-faces-from-images-using-opencv-python/