Facial Recognition Biometric System

October 1st, 2021

https://github.com/EladGashri/Facial_Recognition_Biometric_System

Elad Gashri
https://www.linkedin.com/in/elad-gashri

Yonatan Jenudi M.D.
https://www.linkedin.com/in/yjenudi

Introduction

The Facial Recognition Biometric System is a final project in Tel Aviv University's Faculty of Engineering. It was developed in collaboration with Amdocs.

In this project we designed and implemented a biometric system that uses a facial recognition algorithm. The algorithm was developed by us from scratch with Python and Pytorch.

In the preprocessing, the system found the pictures and names of the people it should train on and detect.
The DataFrame that has been built in the preprocessing is used to create the database in MongoDB. After that, we have a reliable database, and the system queries the people it could train on and divides it into two sets (train and validation).
The two sets are moved into a pre-trained artificial neural network (ANN) that was accommodated according to the dataset's needs.

The system uses few methods to avoid overfitting, like augmentation and a weighted random sampler. The system also uses some optimization methods like "ReduceLROnPlateau."
After the learning process, the system updates a score (probability) for each person in the database. That's the way that the user of the system can keep track of the system performance.

The system was designed to be a real-time attendance system for a large corporation.
The system interacts with several cameras in different locations, where employees should come to be identified. If an employee has been identified by the facial recognition algorithm, his attendance will be registered at the database. Otherwise, the employee could be identified manually using his personal information.

The facial identification takes place in one location: The facial recognition algorithm stand. There can be many camera stands, separate from the facial recognition algorithm stand.
All the camera stands interact with the algorithm stand using the UDP protocol. If a face has been detected by the camera, the image is sent to the algorithm stand. The Facial recognition algorithm is activated and the result is sent back to the camera stand. The Live Feed was implemented using multithreading in Java.

One of the main purposes of this system is to function as an attendance system. The database holds details about the employees, their attendance and their images.

The system can be interacted with by sending HTTP requests to its REST API. A standard employee can only receive information about himself. An admin can receive information about all the employees. The CTO can also use the facial recognition algorithm to identify an image and to retrain the algorithm. The REST API was implemented with Java, Spring Boot and Spring Security.

Deployment Instructions

Facial recognition algorithm stand

Follow these instructions in order to deploy the Facial Recognition Biometric System with your employee's images and details.

System requirements:

- Python version >= 3.0.
- Java version >= 15.0.
- Maven.
- MongoDB.
- Windows OS or Linux OS (for Linux in the paths replace the backslash (\) with a forward slash (/)).

1. Download the code from https://github.com/EladGashri/Facial_Recognition_Biometric_System. Name the root directory biometric_system_github.

2. Set up a Python virtual environment according to the requirements.txt file in biometric_system_github\python.

3. Set the algorithm stand IP address and port in the LiveFeedManager class in LIVE_FEED_SERVER_ADDRESS and LIVE_FEED_SERVER_PORT. The LiveFeedManager class is in biometric_system_github\java\com\biometricsystem\livefeed.

4. Have a dataset directory of your employees' images. The directory's path should be biometric_system_github\resources\static\images\dataset. In that directory, there should be 3 directories, for the 3 employees types:

   - standard
   - admin
   - cto

   The system is designed for there to be many admins and one CTO. Nevertheless, the only requirement is for there to be at least one CTO (who can also act as an Admin).

   Each employee should have his images directory within one of the above directories, according to his employee type. The employee's directory should be the employee name, separated by an underscore (_).
   In order to be included in the model each employee should have a minimal amount of images, defined in the Employee classes in MINIMUM_NUMBER_OF_IMAGES_FOR_MODEL. The Employee classes are in biometric_system_github\java\com\biometricsystem\entity\employee and in biometric_system_github\python

5. Set the current directory as biometric_system_github.

6. Run the command:
   python python\main.py

7. Set your database IP address in the Database classes in DATABASE_ADDRESS. The Database classes are in biometric_system_github\java\com\biometricsystem\database and in biometric_system_github\python.

8. Run the command:
   python python\database.py

9. Build a Maven project according to the pom.xml file. Name Maven project root directory as biometric_system_maven.

10. Move the directory biometric_system_github\python to biometric_system_maven\src\main.

11. Replace the directory biometric_system_maven\src\main\resources with biometric_system_github\resources.

12. Set the current directory as biometric_system_maven.

13. Run the command:
    java target\classes\com\biometricsystem\main\BiometricSystem

Camera stand

Follow these instructions in order to deploy a camera stand:

System requirements:

- Java version >= 15.0.
- Maven.
- Web camera.
- Windows OS or Linux OS (for Linux in the paths replace the backslash (\) with a forward slash (/)).

1. Download the code from https://github.com/EladGashri/Facial_Recognition_Biometric_System. Name the root directory biometric_system_github.

2. Set the algorithm stand IP address and port in the LiveFeedManager class in LIVE_FEED_SERVER_ADDRESS and LIVE_FEED_SERVER_PORT. The LiveFeedManager class is in biometric_system_github\java\com\biometricsystem\livefeed.

3. Build a Maven project according to the pom.xml file. Name Maven project root directory as biometric_system_maven.

4. Set the current directory as biometric_system_maven.

5. Run the command:
   java target\classes\com\biometricsystem\livefeed\client\LiveFeedClientMain

Development

The development phases are as follows and will be explained in details next:

1. System design, tools selection and dataset selection.
2. Data engineering and preprocessing.
3. Database design and implementation.
4. Facial recognition algorithm design and implementation.
5. REST API design and implementation.
6. Live feed design and implementation.

Tools

The dataset chosen for this project is "Labeled Faces in the Wild" which includes 13,233 images of 5,749 people. The people in the dataset represent the employees in the company. The employees were randomly distributed to 4 branches. 99% of the employees were labeled as standard employees, 1% of the employees were labeled as admins and 1 admin was chosen to be the CTO.

The preprocessing on the dataset was done with Python and Python's main data analysis and machine learning libraries: Numpy, Pandas and OpenCV. For the augmentation process we used the Albumentations library.

The facial recognition algorithm was developed with Python and PyTorch. The algorithm was embedded in Java using "Deep Java Library" (DJL).

The database was implemented with MongoDB.

The Live Feed was developed with Java. The GUI that interacts with the employee in the Live Feed was developed with the Java library Swing.

The REST API was developed with Java, Spring Boot and Spring Security.

Data Engineering and Preprocessing

The purpose of preprocessing in our project is to create the database and transform the data into more prepared data to train our model.

The raw dataset contains a folder for each person. Each folder includes a few pictures of a unique person. Therefore, the number of images each person has can differ from one to a couple of dozens.

In the first part of the preprocessing, the system creates a DataFrame (a 2 dimensional Panda's data structure) while iterating over the raw data folders and pictures. The system ignores files that are not images by selecting only files with the postfix of 'jpg,' 'jpeg', and 'png'.

The first DataFrame has only two columns, one for the name of the person and the second for the paths of the images in the machine. Therefore, each person has several paths (the number of pictures in the raw data).

The second part of the preprocessing is to check if we can use each image for training.

We set three criteria for this matter:

1. The system can produce the "face indexes" - two pixels in the picture that mark the position of the face in the image.
2. The "face indexes" width and height cannot be less than zero, which means that the face is out of the picture.
3. For each person, the system takes a maximum number of 10 pictures. We do so to avoid overfitting our model over one person with many pictures against people with fewer pictures.

According to these criteria, the system iterates over each image, identifies where the face is located, and saves the "face indexes" in the DataFrame.

The system also keeps track of the number of valid pictures (those that could produce "face indexes") and save this number for later use. This number is recorded for each employee in his database document in the “number of images” field.
The system keeps only the paths that contain valid pictures because we cannot use the other images for our model.

The system finds the face indexes with the open-source python library called MTCNN.

The MTCNN library returns four numbers for two points in the image.
The upper-left point (x,y) of the face.
The lower-right point (h,w) of the face.

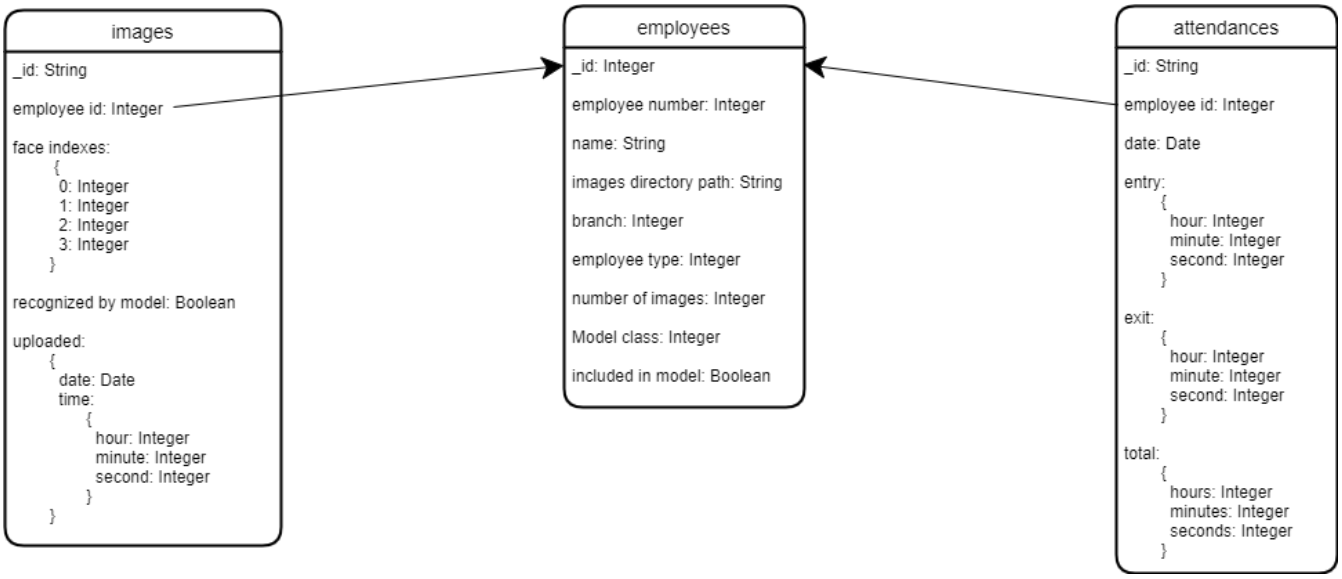
Database

The data that is inserted to the database is the data in the DataFrame that was created in the preprocessing stage.

The database was designed as a NoSQL document database and was implemented using MongoDB.

The database includes 3 collections: employees, images and attendances.

Each document in the images and attendances collections includes a reference to a document in the employees collection: employee id. This means that the relationship between the employees collection with both the images collection and the attendances collection is one-to-many. Each employee can have many images and many attendances.



Facial Recognition Algorithm

Training and model

The system creates the train set and the validation set before the training process.
To do so, the system queries from the database only the persons with four or more images with valid indexes. Then, it creates a map (a dictionary) that connects the person ID and the "class number" they get for the model, so it updates the new accuracy score for this specific ID after the training.
People who do not have four or more images will need to add face images to the database before the model can detect them.

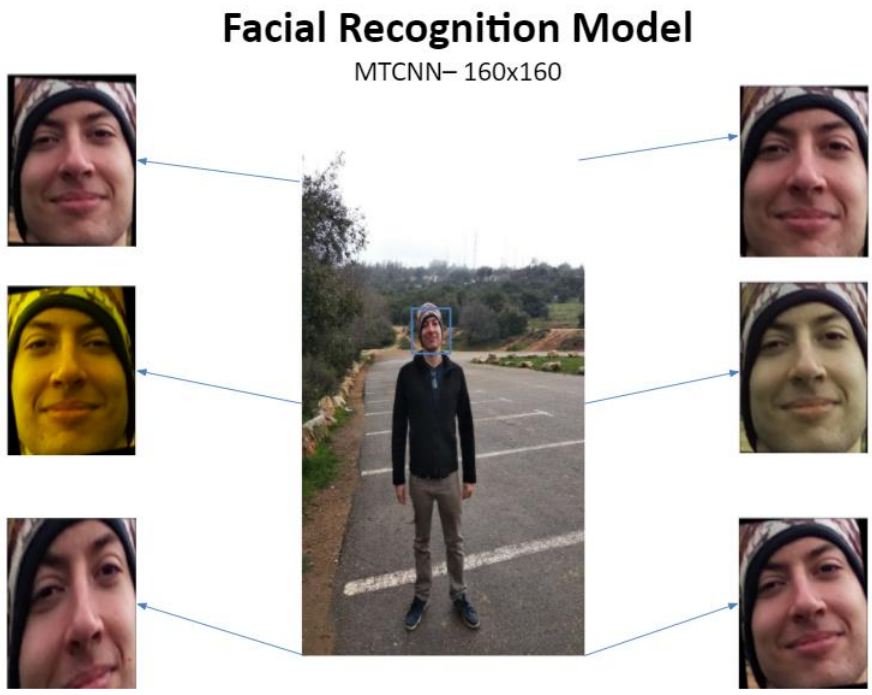
Pytorch dataset

We use the PyTorch dataset module to prepare the images before they are getting fed to the model.
The purposes of the PyTorch dataset module:

1. Cut the image to contain only the face. We used the indexes that the MTCNN library set in the preprocessing step.
2. Resize all images into a fixed size of 160x160. This resolution was used to train the InceptionResnetV1 model, which we use for our project.
3. Normalize all images to have zero mean and one standard deviation.
4. Convert the images into a Tensor data type.
5. Some images indexes set by MTCNN library were out of the size of the image (this occurs when part of the face is outside of the image). This issue was addressed for each image when passing through the Pytorch dataset module.
6. Only images within the train set were sent into our augmentation function to avoid overfitting. This step also occurs with the PyTorch dataset.

Augmentation

We assume that we do not have lots of images for each person we need to detect. Therefore, the system enlarges the data with augmentation. We use the Albumentations library for this purpose.
An example for the cropping and augmentation processes can be seen in the following figure:



PyTorch WeightedRandomSampler

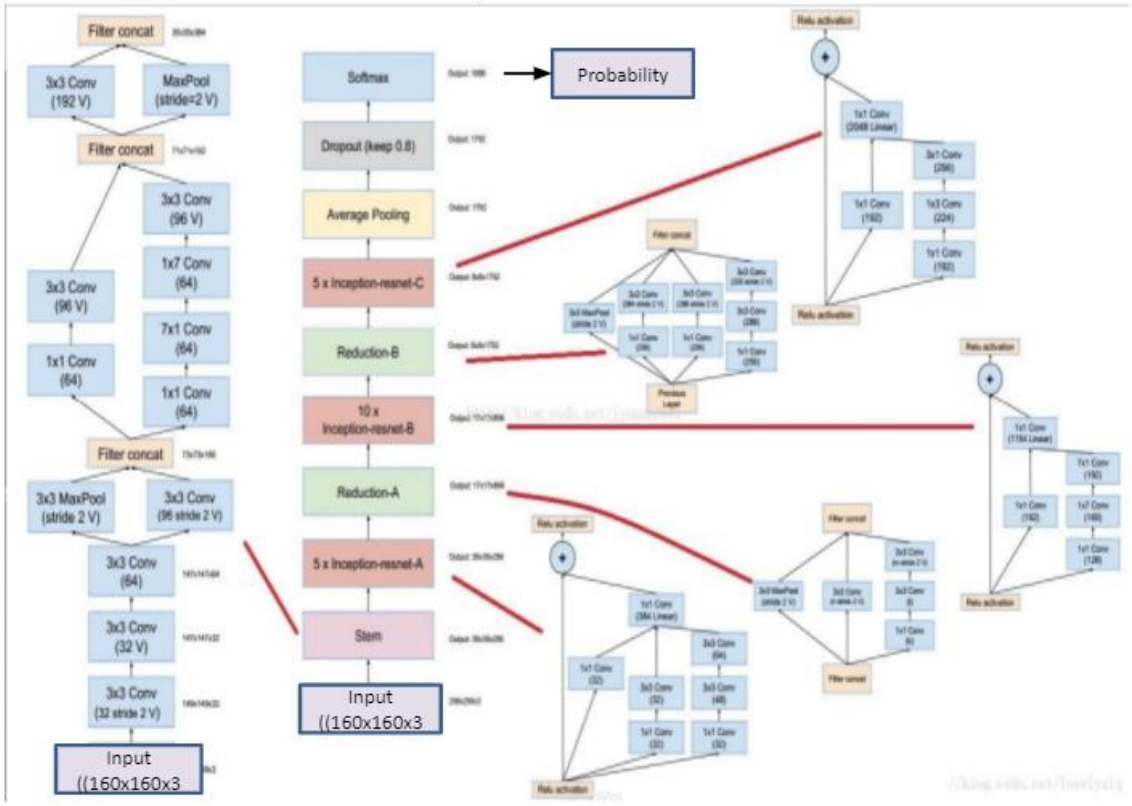
Because some people have more pictures than others, we want to avoid a state where our model learns to detect one person (the one who has more pictures) better than others (overfitting over one class).
We address this issue by using the PyTorch WeightedRandomSampler module.
Before each training process, we create a WeightedRandomSampler that is accountable for displaying the images in the same ratio regardless of the number of images that each person has to the net.

The model

We use the pre-trained inception Resnet model that was trained on VGGFace2 data.
We change the classifier part of the model according to the number of persons or classes we want to classify.
In addition to the pre-trained model, we also add another dropout layer and a linear layer.
The pre-trained model architecture can be seen in the following figure:

Facial Recognition Model - Architecture

Inception-ResNet-v1



The training process

Because our model consists of lots of parameters, we recommend training it with GPUs instead of CPUs..
After the training of N epochs, we save the model weights for detection or further training.
We use Adam optimizer with a learning rate of 0.0001 and l2 regularization of 0.001 for 20 epochs.
We train our model mainly on the GPU of Google Colab.
We deal with a multi-class challenge, so we use the CrossEntropyLoss function to fit this classification.
The batch size should be picked as a result of the number and the type of processor that the model will be trained on.

To optimize our result, we use the PyTorch package - ReduceLROnPlateau.
This function can be adjusted for better results. For example, we set it to reduce the learning rate by a factor of 0.1 if the validation loss does not improve after seven epochs. This helps the model to converge into a lower minimum point.

After the training process, we update the score for each person trained with the probability score gained by the Softmax function from the model.

Algorithm embedment in the Java code and the REST API

The algorithm is automatically being trained every 24 hours. This process is implemented with Java's Executors class which allows scheduling a thread that runs at a fixed rate. The thread that runs at a fixed rate acts as an HTTP client: it sends an HTTP POST request to the REST API's URI /model/training. The request is being sent as the CTO, because only the CTO has the credentials to train the algorithm.

The training process takes place by running a python code that queries all the images in the database and inserts them to the algorithm training code. After the training the new algorithm file will be one used for identifications.

The algorithm was written with Python and PyTorch. The algorithm was embedded in the Live Feed and REST API (which were written with Java) using "Deep Java Library" (DJL). The usage of DJL enables us to run the algorithm in Java without the need to create a new process to run the Python code. The algorithm is read from a TorchScript, a representation of a PyTorch model which is independent from the Python programming language.

REST API

A full documentation of the Facial Recognition Biometric System REST API with detailed examples for every endpoint can be found at the following Postman collection document:

https://documenter.getpostman.com/view/14799541/UUy1emcF

The Facial Recognition Biometric System REST API was developed with Java, Spring Boot and Spring Security.

The API allows performing CRUD operations on the system's database, receiving an attendance report for each employee in a chosen date range and interacting with the facial recognition algorithm.

A standard employee can only access data about themselves from the API. An admin can access data about all the employees. The CTO can also get a prediction from the facial recognition algorithm and train the algorithm.

The API was designed with the 3 layer architecture. Each endpoint was implemented with 3 classes:

1. **Controller** - the presentation layer. It is responsible for correctly routing each HTTP request to the correct Service method and returning the correct information and HTTP status codes.

2. **Service** - the business logic layer.

3. **Repository** - the data access layer. It includes access to the database for retrieving, inserting, updating and deleting data.

The Spring framework uses dependency injection in order to inject every instance of those classes to the classes that use them. This allows objects to not have to initialize the classes they use, but instead to get them in advance, and by doing so we achieve loosely coupled classes.

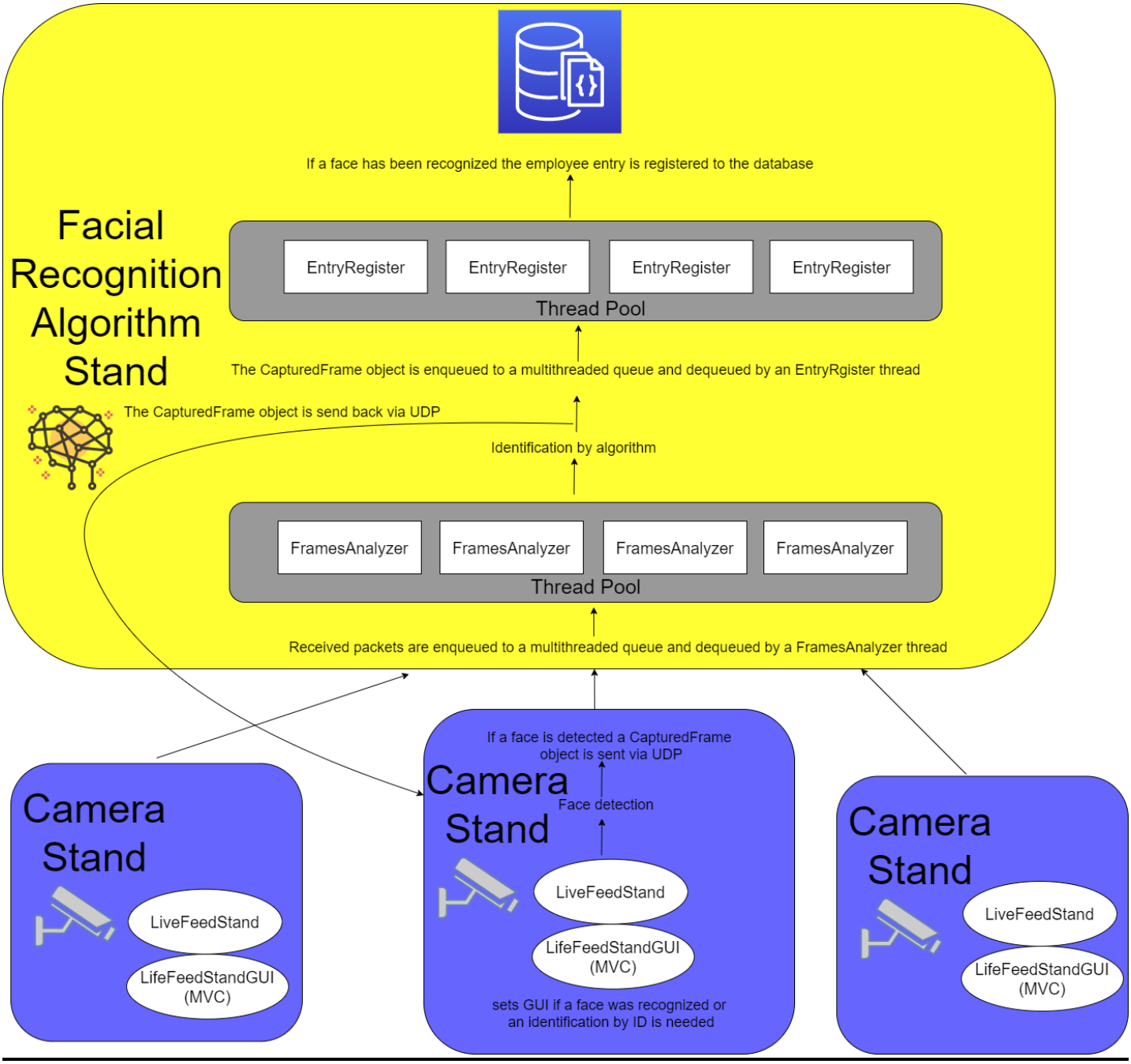
Using dependency injection implements the dependency inversion principle in SOLID. This occurs because there is no dependency on the class implementation. The class that uses it is only dependent on the abstract class. The class that takes the injection doesn't know which implementation it is being injected with, it is only aware of what the interface of that class is.

The API uses JSON Web Token (JWT) for authentication. After authentication the employee received a token. In the next requests the employee will send the token, which will be used for authorization. The token is a string which holds information about the employee. The token can be decoded using a secret key that exists only on the server.

The usage of a JWT for authentication instead of a session allows preserving the stateless principle in REST. The stateless principle in REST is preserved because by using a JWT every request is independent from any other request.

The stateless principle is not preserved with a session authentication. The session's details need to be saved on the server (in the memory or in the database). In every request the client sends a cookie that is used as an id for the session. The fact that the session's details are being saved on the server violates the stateless principle that states the client and server must not save details about each other.

Live Feed


If a face has been recognized the employee entry is registered to the database
Facial Recognition Algorithm Stand
EntryRegister
EntryRegister
EntryRegister
EntryRegister
Thread Pool
The CapturedFrame object is enqueued to a multithreaded queue and dequeued by an EntryRgister thread
The CapturedFrame object is send back via UDP
Identification by algorithm
FramesAnalyzer
FramesAnalyzer
FramesAnalyzer
FramesAnalyzer
Thread Pool
Received packets are enqueued to a multithreaded queue and dequeued by a FramesAnalyzer thread
Camera Stand
LiveFeedStand
LifeFeedStandGUI (MVC)
If a face is detected a CapturedFrame object is sent via UDP
Face detection
Camera Stand
LiveFeedStand
LifeFeedStandGUI (MVC)
sets GUI if a face was recognized or an identification by ID is needed
Camera Stand
LiveFeedStand
LifeFeedStandGUI (MVC)

The live feed purpose is to be the main feature of the real-time processing in the system. The live feed works in collaboration with the facial recognition algorithm to identify employees as they come to the office and to register their attendances. The live feed is implemented with multithreading in Java. The GUI is written with Java library Swing.

The code structure of the live feed can be simplified to 2 parts:

1. **Camera stand**.

2. **Facial recognition algorithm stand**.

There could be multiple camera stands and there is always 1 algorithm stand. This is an implementation of a client-server architecture. The clients are the camera stands and the server is the algorithm stand. The clients send identification requests to the server, which returns the identification results. The communication is based on the UDP protocol. UDP was chosen because it is ideal for missions such as live feed, where we require low-latency and can tolerate a packege loss, because many frames are taken.

The camera stand code was designed for there to be minimal hardware requirements. This stand should be equipped with a camera, an internet connection and a device that can perform simple face detection.

When an employee approaches the camera his face will be detected. A CapturedFrame object will be initialized with his frame. CapturedFrame is the class that is consumed by the algorithm. The CapturedFrame object is then send via UDP to the algorithm stand.

The algorithm stand was designed to be multithreaded in order to be able to handle requests from multiple camera stands simultaneously. The Facial recognition algorithm includes 2 thread pools:

1. **FramesAnalyzer thread pool**.

2. **EntryRegister thread pool**.

When the UDP packets arrive to the algorithm stand they are enqueued to a multithreaded queue. FramesAnalyzer threads dequeue them. The FramesAnalyzer thread performs facial identification using the algorithm, sets the results to the object (face either recognized or not) and sends the CapturedFrame object back to the camera stand it came from.

The FramesAnalyzer thread also enqueues the CapturedFrame object to another multithreaded queue from which EntryRegister thread dequeue. The EntryRegister thread checks if a face has been identified, and if so, the thread registers the employee attendance to the database.

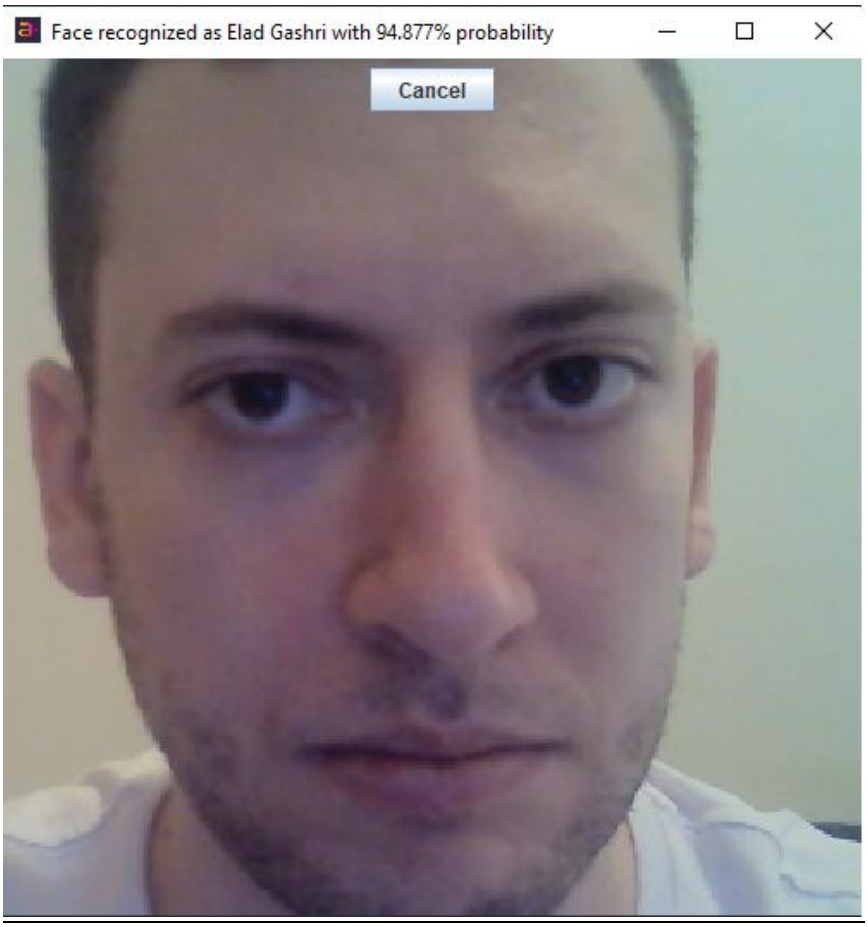
The camera stand code also includes a GUI for displaying the identification result. The GUI is implemented with the MVC architecture. This architecture includes:

1. **Controller** – receives the UDP packet back from the algorithm stand. The Controller Facilitates the interaction between the model and the view by sending the CapturedFrame object first to the Model and then from the Model to the View.

2. **Model** – checks the identification result of the image and according to the result sets the data and sends it to the controller.

3. **View** – initializes the graphics according to the data it was sent from the controller. If the face has been recognized (by passing the recognition threshold threshold) the image is presented with the employee name and the recognition probability. If the face was not recognized the employee can manually identify by entering his ID and employee number. If the employee entered his correct information his entry is registered and the image is saved, for future algorithm training.

<u>GUI in response to a face recognized:</u>



GUI in response to a face not recognized: