STGI Technologies Consulting Pvt Ltd

Hackathon 2024

Problem Statement: User KYC Verification Platform

Documentation

Team Name: Byte Me

Arav Gupta Armaan Verma

Lakshya Verma Shauray Dhingra

Sarthak Kapila

Acknowledgement

We would like to extend our sincere thanks to STGI Company for organizing and hosting this incredible 24-hour hackathon experience. The challenge provided us with the opportunity to test our skills, collaborate with peers, and push the boundaries of innovation within a limited timeframe. Your support, resources, and commitment to fostering a creative and competitive environment have been invaluable in helping us grow both technically and personally.

Thank you for this unforgettable experience!

User KYC Verification Platform

Overview

The User KYC (Know Your Customer) Verification Platform ensures user authenticity by leveraging advanced facial recognition and image verification techniques. The platform captures a live image of the user, verifies it against an ID document, and checks a second image against a large-scale database for matching purposes.

The platform is built using the **DeepFace** library, OpenCV, Dlib, and several machine learning models for accurate face detection, verification, and liveness detection.

Key Features:

1. **Liveness Detection**: Ensures that the image captured via webcam is live and not a spoofed attempt using pre-recorded images or videos.
2. **Document Verification**: Matches the live image against the face in the user-uploaded ID document to verify identity.
3. **Database Image Matching**: Compares a secondary image of the user against a large dataset for potential matches.

Table of Contents:

Problem Statement

The task is to design a comprehensive **KYC (Know Your Customer) verification platform** that ensures user authenticity through advanced image verification techniques. The platform needs to capture a live image of the user, verify it against a provided ID document, and compare a second image against a large database to check for potential matches.

Workflow Overview:

1. **Capture Live Image**:
    - The platform must take a live picture of the user to confirm they are physically present during the verification process.
2. **Upload Documents**:
    - The user will upload two images:
        - An **ID document** (e.g., passport, driver's license) for facial verification.
        - A **second image** for database comparison, which can be a random photo of the user.
3. **Verification Process**:
    - The system will:
        - **Compare the live image** captured from the webcam with the face in the ID document to verify the user's identity.
        - **Search the second image** against a large image database to find the top-k similar matches and return respective similarity scores.

Key Components:

1. **Liveness Detection**:
    - Ensure the live image is genuine and not a spoof (e.g., a photo of a photo). This can include checks such as **blink detection**, **head movements**, or other creative real-time interaction measures.
2. **Flexible Image Formats**:
    - Accept various image formats for both the live and uploaded images, ensuring accurate **facial recognition** despite potential variations in lighting, angles, and accessories like glasses or hats.
3. **Efficient Database Search**:

- The platform should be able to **search efficiently** through millions of images in the database. This can be achieved using **algorithms** that ensure fast and reliable comparisons (e.g., cosine similarity, KNN). Use an open-source image dataset for testing and verification of the approach.

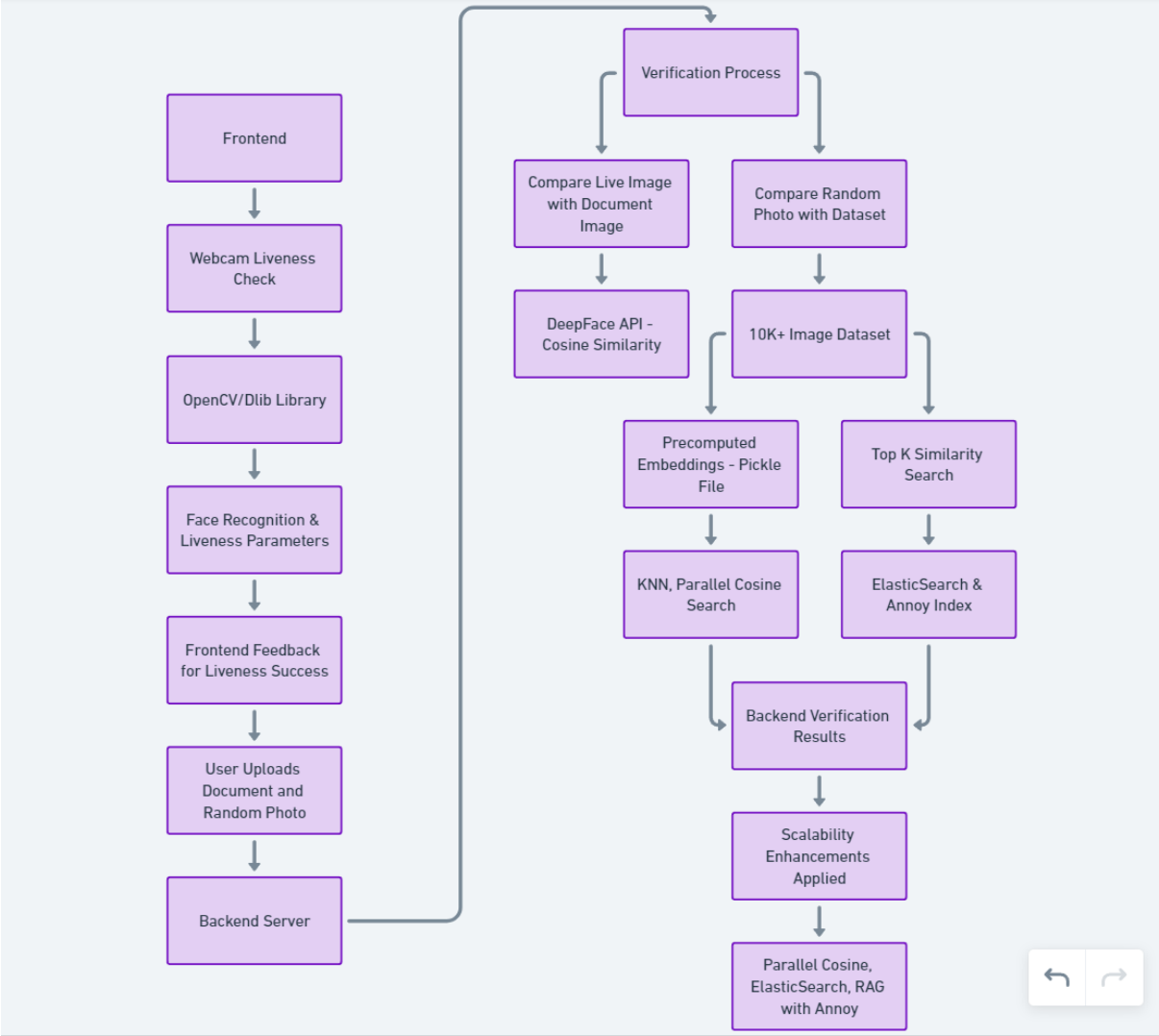
4. **Scalability**:

- The platform must be scalable to handle a **high volume of users** without performance degradation. This includes optimizing for large datasets and ensuring the system performs efficiently under load.

System Architecture

Main Components:

1. **Liveness Detection Module**: Verifies user presence through webcam by checking for blinking, head movement, and gesture recognition.
2. **Document Verification Module**: Extracts the face from the user's ID document and compares it with the live image.
3. **Database Matching Module**: Compares the user's second image with a large dataset of images, using cosine similarity and k-NN to find the top k matches.



Workflow Overview

1. **Capture Live Image**:
    - The system captures a live image of the user through the webcam. It performs three key checks for liveness detection:
    - **Blinking Detection**: Tracks eye movement for blinking.
    - **Head Movement**: Tracks the user's chin movement.
    - **On-Prompt Gesture**: Asks the user to show a V-sign gesture.
2. **Document Upload**:
    - The user uploads an ID document and a second random image for the database comparison.
3. **Verification Process**:
    - The live image is compared with the ID document using DeepFace's facial recognition capabilities.
    - The second image is compared against a dataset of 10,000+ images using optimized image comparison algorithms like cosine similarity.
4. **Verification Output**:
    - After passing the verification checks, the user is either approved or flagged.

Key Components

1. Liveness Detection:

The platform uses OpenCV and dlib libraries to perform liveness detection. The following methods are implemented:

- **Blink Detection**: Uses eye aspect ratio (EAR) to detect natural blinking behavior.
- **Head Movement Detection**: Tracks head movements by monitoring the chin's movement.
- **On-Prompt Gesture Detection**: Prompts users to perform a physical gesture like showing a V-sign, verifying real-time interaction.

2. Document Verification:

- Extracts the user's face from the uploaded ID document and compares it with the live image using the **DeepFace** library's verify function, which applies state-of-the-art facial recognition models.
- **Similarity Metric**: Uses cosine similarity for verification.

3. Database Image Matching:

- The user’s second image is compared against a dataset of 10,000+ images to find the top k matches.
- Uses **cosine similarity** and **K-nearest neighbors (KNN)** for faster performance.

Technology Stack

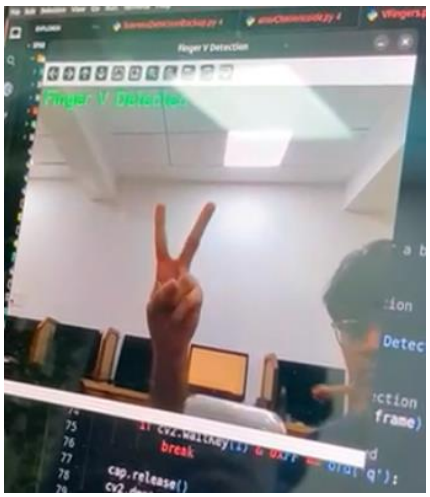
Component	Technology Stack
Frontend	TensorFlow.js, JavaScript, HTML, CSS
Backend	Python (Flask), OpenCV, Dlib, DeepFace
Database	Elasticsearch, Annoy for Approximate Nearest Neighbor search
Facial Recognition	DeepFace (VGG-Face, ArcFace, FaceNet)
Model Training	TensorFlow, PyTorch

1. Liveness Detection:

Liveness detection is the process of ensuring that the user interacting with the system is physically present and not using pre-recorded photos or videos. For this purpose, we implemented several techniques using **OpenCV** and **Dlib** to check user interaction and ensure the image captured via the webcam is authentic.

Techniques Used:

- **Blink Detection**:
    - We used facial landmarks to track the user's eye movement, calculating the **Eye Aspect Ratio (EAR)**, which changes when a person blinks.
    - A certain threshold is defined, and if the EAR falls below this threshold, it is considered a blink. The system tracks whether the user blinked within a given time frame, ensuring the user is live.
- **Head Movements**:
    - By tracking the position of the chin using facial landmarks, we detect subtle head movements.
    - A **chin movement threshold** is defined to measure if the user moves their head up, down, or sideways within a specific time frame. This helps verify real-time movement, which static images or pre-recorded videos can't replicate.
- **On-Prompt Gesture Detection**:
    - As an additional security measure, the system prompts the user to perform a specific gesture, such as showing a V-sign with two fingers.
    - This on-prompt movement helps ensure that the user is present and interacting live with the system.



Challenges:

- **Depth Detection**: To prevent spoofing using pre-recorded videos, we intended to use depth detection to measure the physical distance of the user from the webcam. However, due to GPU constraints, we were unable to implement this feature in the current version.
- **API and Sensitivity**: The **API calculates EAR** for blink detection, and the **chin movement sensitivity** has been fine-tuned to optimal values to ensure accurate results.

2. Document Upload and Cropping for Verification:

In this stage, the user uploads two images:

1. A document (e.g., passport, driver's license) containing a photo of themselves.
2. A second random image of themselves for database comparison.

Future Image Processing:

- After uploading the document, we automatically crop the face from the ID document to extract the photo for comparison. This involves detecting the face region in the document using OpenCV and facial landmark detection.

Features:

- The system prepares the document image for facial recognition, cropping only the face for a clean comparison with the live webcam image captured during the first step.
- The second image (random image) is stored for the database comparison step to ensure further verification.

3. Verification Process

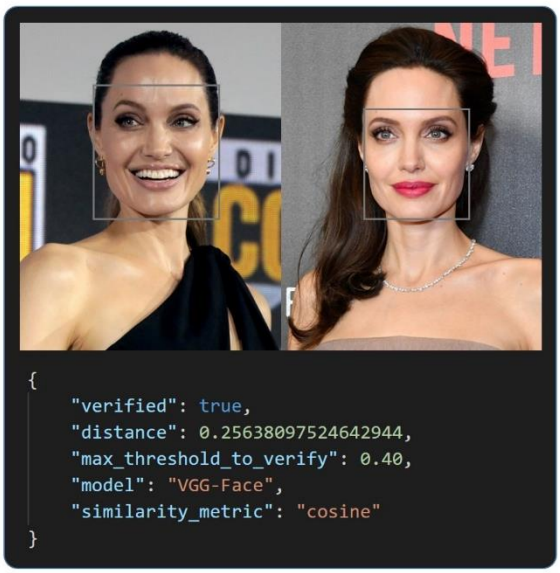
The verification process involves two verticals:

1. **Live Image Verification Against Document Image**:
    - We utilize the **DeepFace** library, which provides pre-trained facial recognition models (such as VGG-Face, FaceNet, and ArcFace) to compare the live image with the ID document photo.
    - **Cosine Similarity** and **Euclidean distance** are used as the metrics for measuring the similarity between the images.
        - Cosine similarity is particularly useful because it focuses on **directional differences** rather than magnitude differences in the facial vectors, which is crucial in face recognition.
        - A **predefined threshold** is set, and if the similarity score between the live image and the document image exceeds this threshold, the identity is considered verified.

    - **Verify** Function of Deepface:

      This function verifies face pairs as same person or different persons. It expects exact image paths as inputs. Passing numpy or base64 encoded images is also welcome. Then, it is going to return a dictionary and you should check just its verified key.

```
result = DeepFace.verify(
  img1_path = "img1.jpg",
  img2_path = "img2.jpg",
)
```



2. **Second Image Verification Against Dataset**:
    - The second random image uploaded by the user is compared against a **large dataset** of faces (10,000+ images).
    - We have extracted **facial embeddings** from each image in the dataset using models like VGG-Face and stored these embeddings in a **pickle file**. This helps in quick comparisons.
    - **Top K Cosine Similarity** is used to find the most similar matches in the dataset. This optimization reduces time complexity from **O(n^2)** to **O(n)** by only scanning the top k most likely matches.

In our solution, we have created a **Flask-based API** to handle efficient similarity searches using **facial embeddings**. The core of the system is powered by the **Annoy library** for fast **approximate nearest neighbor (KNN)** searches, which significantly improves performance when dealing with large datasets.

Implementation:

1. **Embedding Loading and Index Creation**: We load the facial embeddings from a **pickle file** and use **Annoy** to index them. Annoy's **angular distance** metric and 10 trees are used to ensure efficient similarity searches. This process ensures that even with large datasets, the search remains performant and lightweight.

```
def load_embeddings():
    # Loading embeddings from pickle and building Annoy index
```

2. **Neighbour Search Endpoint** (/find_neighbors): We created an endpoint to handle search requests where the user provides a facial embedding and the

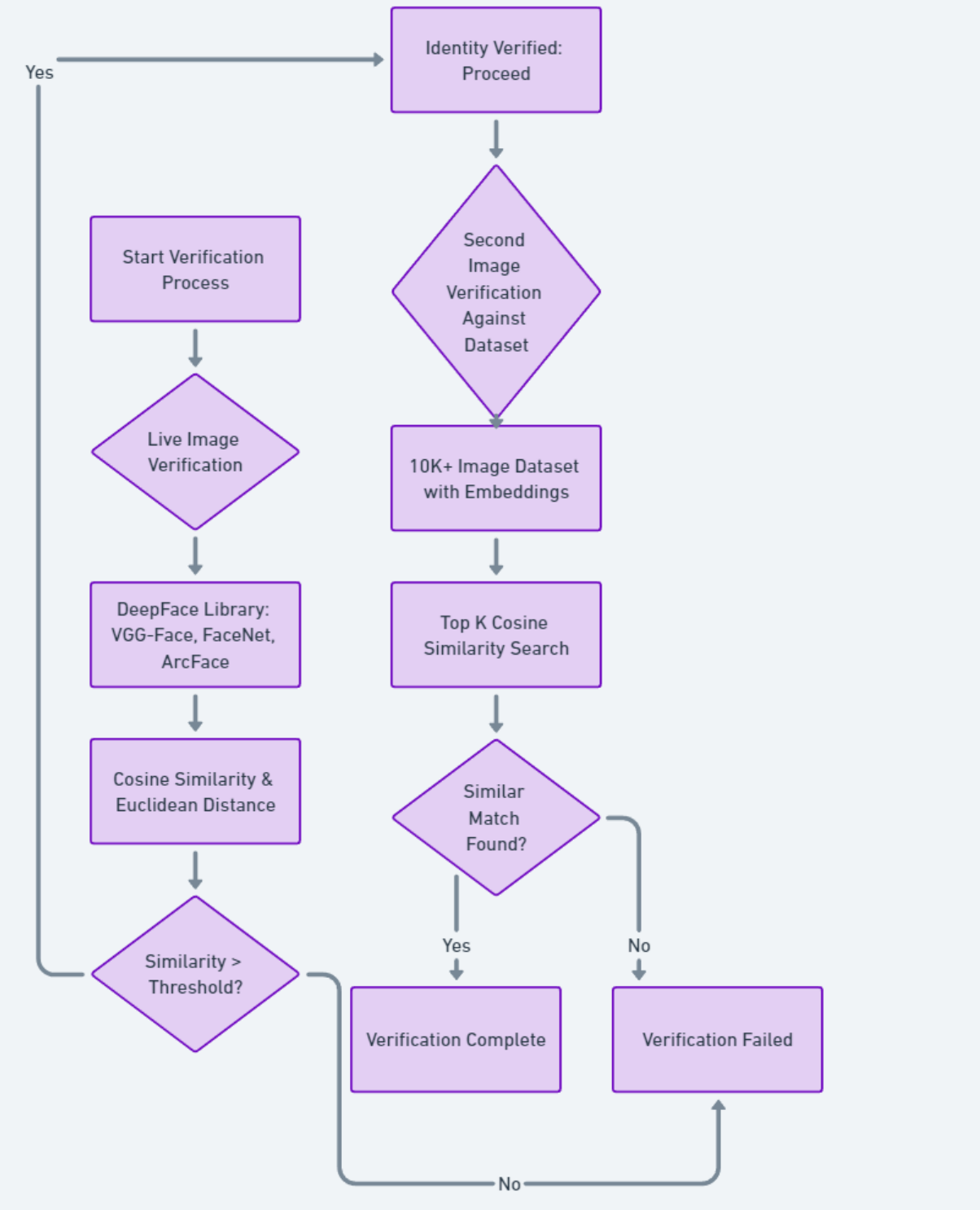
number of nearest neighbors (k) they wish to find. This endpoint validates the input embedding and runs the search using the **Annoy index**.

```
@app.route('/find_neighbors', methods=['POST'])
def find_neighbors():
    # Searches for top-k neighbors using Annoy index
```

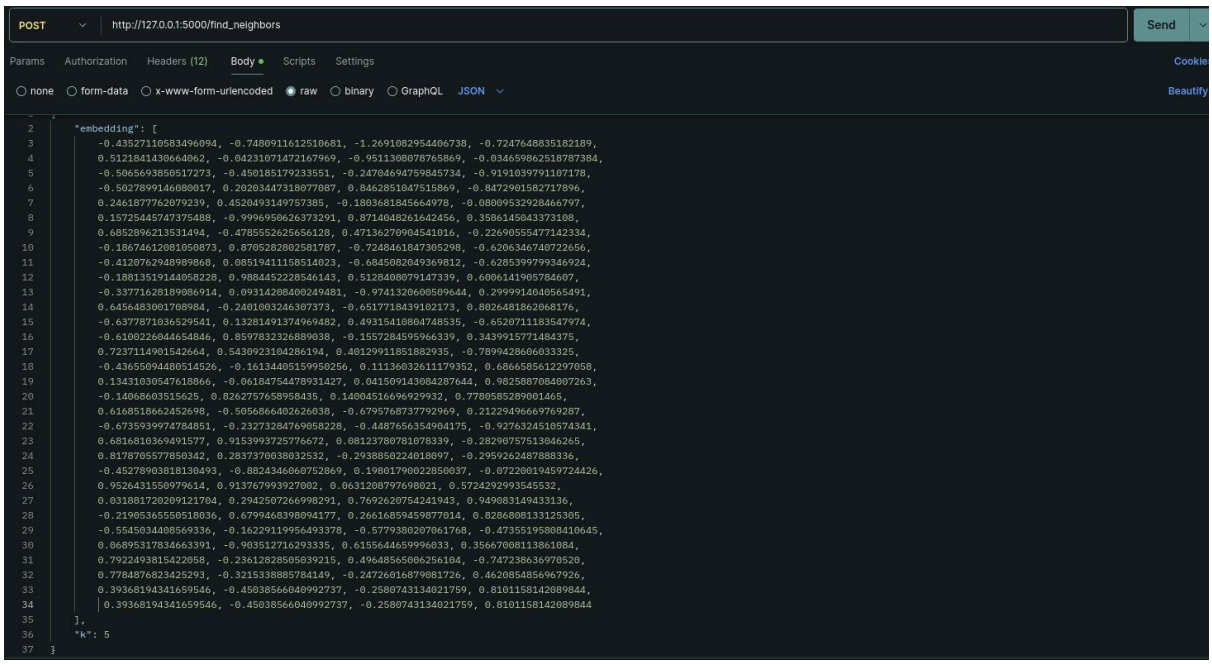
3. **Health Check Endpoint**: To monitor the API's health and ensure the index is properly loaded, we implemented a **health check endpoint** that provides a simple status update.

```
@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({"status": "healthy", "index_loaded": index is not None})
```

4. **Logging and Error Handling**: We integrated robust **logging mechanisms** using Python's logging module to keep track of API activity, embedding load status, and errors. This helps with troubleshooting and monitoring the overall system performance in real-time.



Pic: workflow of verification processes for task3



Pic: Embeddings screenshot of api tested out of 10 Thousand other enteries.

Pic: Response using KNN based which was calculated in 3 seconds.

In our second approach, we developed a **Cosine Similarity Microservice** using Flask to handle fast vector comparisons. This microservice allows for real-time calculation of cosine similarity between two vectors, which is essential for tasks such as facial embedding comparison in our KYC verification system.

Key Features:

1. **Cosine Similarity Calculation**: The microservice receives two vectors via a **POST** request, and computes their **cosine similarity** using the sklearn library. This method is effective in determining the similarity between two high-dimensional vectors, commonly used in facial recognition.

```python
@app.route('/cosine_similarity', methods=['POST'])
def calculate_cosine_similarity():
    vector1 = np.array(request.json['vector1']).reshape(1, -1)
    vector2 = np.array(request.json['vector2']).reshape(1, -1)

    similarity = cosine_similarity(vector1, vector2)[0][0]
    return jsonify({"similarity": float(similarity)})
```

2. **RESTful API**: The service exposes an endpoint /cosine_similarity that accepts a **POST** request with two vectors in JSON format. The result is returned as a similarity score, making it easy to integrate this service with larger systems that need to compare embeddings.
3. **Lightweight and Scalable**: This microservice is highly **lightweight**, as it relies only on core libraries like Flask, NumPy, and sklearn. It is designed to be easily deployable and can handle multiple requests for fast cosine similarity computations, ensuring scalability for high-demand environments.

By integrating the **Cosine Similarity Microservice**, the **response time was significantly reduced to 0.9 seconds**, compared to the previous 3 seconds, while maintaining the same high accuracy in facial embedding comparisons. The efficiency of cosine similarity for measuring vector distances allows us to compute results faster, even with high-dimensional data, without sacrificing precision.

Additional Benefits:

1. **Lightweight and Fast**: The microservice is highly lightweight and optimized for performance, allowing it to handle large volumes of requests with minimal overhead.
2. **Scalable Design**: The service can easily scale horizontally by deploying multiple instances, each handling vector comparison requests in parallel, ensuring the system can accommodate high traffic and large datasets.
3. **Real-Time Applications**: This approach is ideal for real-time systems where rapid response times are critical, such as **live identity verification** or **fraud detection** in our KYC platform.
4. **Low Computational Overhead**: The use of **cosine similarity** keeps the computational overhead low, even when handling high-dimensional vectors, making it an efficient solution for facial recognition tasks.

We have prototyped a **Flask-based Elasticsearch Microservice** to handle text-based search queries efficiently. This service leverages Elasticsearch's powerful indexing and search capabilities to return matching documents from large datasets in near real-time.

Code Overview:

We designed the microservice to perform text search queries using the following approach:

1. **Elasticsearch Client Initialization**: The service connects to a local **Elasticsearch instance** running on http://localhost:9200.

```python
es = Elasticsearch(["http://localhost:9200"])
```

2. **Search Endpoint** (/search): The endpoint accepts a **POST request** containing two key parameters in the request body:
    - **query**: The text to search for.
    - **index**: The specific Elasticsearch index (similar to a database table) where the search is performed.

```
query = request.json['query']
index = request.json['index']
```

3. **Executing the Match Query**: Using the **Elasticsearch client**, a **match query** is run on the specified index to search for the provided query text within the "content" field.

```
result = es.search(index=index, body={
    "query": {
        "match": {
            "content": query
        }
    }
})
```

4. **Returning Search Results**: The service returns the matching documents (hits) in **JSON format**, allowing the results to be easily integrated with other parts of the system.

```
return jsonify(result['hits']['hits'])
```

Performance Comparison with KNN and Cosine Models:

To estimate the time efficiency of this microservice, we compared its performance with the **KNN with Cosine Similarity** model used in our vector-based facial recognition system.

1. **KNN with Cosine Similarity**:
    - **Usage**: Ideal for vector-based similarity searches, such as comparing facial embeddings.
    - **Time Complexity**: Depends on the number of vectors and dimensions. Optimized using **Top-K Cosine Similarity** and **pre-extracted embeddings**.
    - **Response Time**: Reduced to **0.9 seconds** in our system, compared to the initial 3 seconds.
2. **Elasticsearch Match Query**:
    - **Usage**: Optimized for **text-based search** in large datasets.
    - **Time Complexity**: Elasticsearch is distributed and performs efficient text indexing and searching.
    - **Response Time**: For simple match queries, Elasticsearch typically returns results in **0.1-1 second**, making it faster for text-based searches compared to vector-based similarity searches.

By integrating the **Elasticsearch Microservice** for text-based searches, we are able to achieve near real-time response times, often **under 1 second**. This complements our **KNN with Cosine Similarity model**, which is used for vector-based searches like facial recognition, with a response time of **0.9 seconds** after optimization.

Together, these two microservices allow our system to handle both **text-based queries** and **vector-based comparisons** efficiently, providing a scalable and lightweight solution for different search scenarios.

Scalability

To make the system scalable and handle large volumes of data, we considered several approaches for optimizing the image comparison process. The system is designed to handle millions of images efficiently, ensuring rapid performance even as the dataset grows.

Scalable Approaches Implemented:

1. **Cosine with K-Nearest Neighbors (KNN)**:
    - This method is used to limit the number of comparisons by focusing on the top k most similar images in the dataset. This drastically reduces the time complexity for database searches.
    - By using **KNN** in combination with **Cosine Similarity**, the system can quickly find the best matches with high accuracy.
2. **Parallel Cosine**:
    - This approach leverages **parallel processing** to speed up the cosine similarity calculation. By dividing the workload across multiple threads or processes, the system can handle more requests and larger datasets without performance bottlenecks.
    - **Parallel Cosine Similarity** improves the response time by executing multiple image comparisons simultaneously.
3. **Non-Parallel Cosine**:
    - As a fallback approach, non-parallel cosine similarity is used in environments where parallel processing resources are limited. Though slower than parallel processing, this method is still efficient due to the use of cosine similarity for optimized matching.
4. **Elasticsearch**:
    - To further enhance search efficiency, we have integrated **Elasticsearch** for indexing and searching through the large dataset. Elasticsearch allows fast querying of the stored facial embeddings and retrieves top-k similar faces in a fraction of the time compared to a traditional database search.
5. **RAG with Annoy**:
    - We experimented with **RAG (Retrieval-Augmented Generation)** combined with **Annoy** (Approximate Nearest Neighbors Oh Yeah) for approximate nearest neighbor searches.
    - Annoy is particularly useful for handling large-scale searches with low latency. It performs vector similarity searches quickly by approximating the nearest neighbors, which reduces computation time significantly when working with massive datasets.

Future Scopes of the KYC Verification Platform

Switch to ResNet-50 for AI Models:

- ResNet-50 offers superior accuracy and robustness in handling complex facial recognition tasks.
- It can manage diverse scenarios like occlusions, poor lighting, and angled images better than MobileNet.

Enhanced Liveness Detection:

- Use **depth mapping** to ensure the user is a 3D object, preventing image or video spoofing.
- Incorporate **micro-expression detection** to identify subtle facial movements for more accurate liveness verification.

Client-Side Processing for Lightweight Application:

- Shift facial recognition and liveness detection to the client-side using **TensorFlow.js** for real-time processing.
- **WebAssembly (WASM)** can optimize performance on the browser, reducing server load and improving privacy.

AI-Based Fake Image Detection:

- Implement **deepfake detection** using machine learning to recognize manipulated or AI-generated images.
- Use **GAN-based techniques** to detect inconsistencies in textures, lighting, or facial movements in fake images.

Multi-Factor Authentication (MFA):

- Add **OTP-based authentication** as a second layer of security to complement facial recognition.
- Incorporate **biometric methods** like fingerprints or voice recognition to further secure the identity verification process.