**Building an Application for Facial Recognition Using Python, OpenCV, Transformers and Qdrant**

[[Plaban Nayak](https://nayakpplaban.medium.com/?source=post_page-----a144871f40d9--------------------------------)](https://nayakpplaban.medium.com/?source=post_page-----a144871f40d9--------------------------------)

[Plaban Nayak](https://nayakpplaban.medium.com/?source=post_page-----a144871f40d9--------------------------------)

·

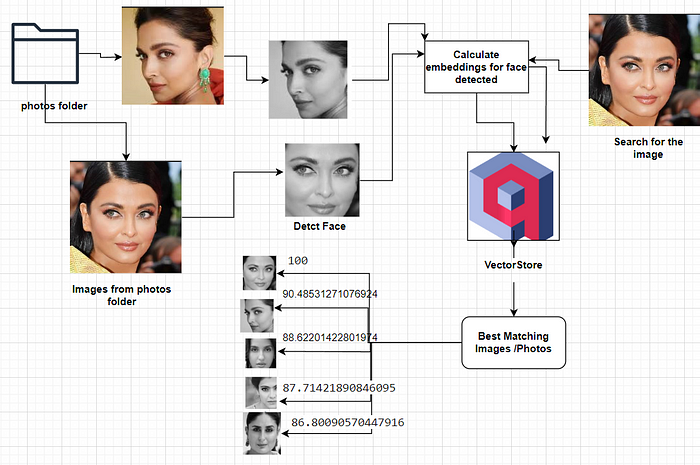
[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F83650c0fa832&operation=register&redirect=https%3A%2F%2Fnayakpplaban.medium.com%2Fbuilding-an-application-for-facial-recognition-using-python-opencv-transformers-and-qdrant-a144871f40d9&user=Plaban+Nayak&userId=83650c0fa832&source=post_page-83650c0fa832----a144871f40d9---------------------post_header-----------)

13 min read

·

Dec 16

124



Face Regonition Application Workflow

**Method 1. Facial Recognition Using Python, OpenCV and Qdrant**

Facial recognition technology has become a ubiquitous force, reshaping industries like security, social media, and smartphone authentication. In this blog, we dive into the captivating realm of facial recognition armed with the formidable trio of Python, OpenCV, image embeddings and Qdrant. Join us on this journey as we unravel the intricacies of creating a robust facial recognition system.

**Part 1: An Introduction to Facial Recognition**

In Part 1, we lay the foundation by delving into the fundamentals of facial recognition technology. Understand the underlying principles, explore its applications, and grasp the significance of Python and OpenCV in our development stack.

**Part 2: Setting Up the Environment**

A crucial step in any project is preparing the development environment. Learn how to seamlessly integrate Python, OpenCV, and Qdrant to create a harmonious ecosystem for our facial recognition system. We provide step-by-step instructions, ensuring you have solid groundwork before moving forward.

**Part 3: Implementation of Facial Recognition Algorithms**

With the groundwork in place, we dive into the core of the project. Explore the intricacies of facial recognition algorithms and witness the magic unfold as we implement them using Python and OpenCV. Uncover the inner workings of face detection, feature extraction, and model training.

**Part 4: Database Integration with Qdrant**

No facial recognition system is complete without a robust database to store and manage facial data efficiently. In the final installment, we guide you through the integration of Qdrant, to enhance the storage and retrieval capabilities of our system. Witness the synergy between Python, OpenCV, and Qdrant as we bring our project to its culmination.

By the end of this blog, you would have gained a comprehensive understanding of facial recognition technology and the practical skills to develop your own system.

**Step-by-Step Implementation**

1. Download all the pictures of interest into a local folder.
2. Identify and extract faces from the pictures.
3. Calculate facial embeddings from the extracted faces.
4. Store these facial embeddings in a Qdrant database.
5. Obtain a colleague’s picture for identification purposes.
6. Match the face with the provided picture.
7. Calculate embeddings for the identified face in the provided picture.
8. Utilize the Qdrant distance function to retrieve the closest matching faces and corresponding photos from the database.

This experiment demonstrates the practical implementation of Python OpenCV and advanced AI technologies in creating a sophisticated Facial recognition / Search Application, showcasing the potential for enhanced user interactions and cognitive responses. Since images are sensitive data ,we do not want to rely on any online service or upload them onto the internet. The entire pipeline defined above is developed to work 100% locally.

**The Technology Stack**

* Qdrant: Vector store for storing image embeddings.
* OpenCV: Detect faces from the images. To “extract” faces from the pictures we used Python, OpenCV, a computer vision tool, and a [*pre-trained Haar Cascade model*](https://www.makeuseof.com/python-detect-faces/)*.*
* imgbeddings: A Python package to generate embedding vectors from images, using [*OpenAI*](https://openai.com/)*’s*robust [*CLIP model*](https://github.com/openai/CLIP) via [*Hugging Face*](https://huggingface.co/)[*transformers*](https://huggingface.co/docs/transformers/index)*.*

**An Overview of OpenCV**

OpenCV, or Open Source Computer Vision Library, is an open-source computer vision and machine learning software library. Originally developed by Intel, OpenCV is now maintained by a community of developers. It provides a wide range of tools and functions for image and video analysis, including various algorithms for image processing, computer vision, and machine learning.

Key features of OpenCV include:

* **Image Processing:** OpenCV offers a plethora of functions for basic and advanced image processing tasks, such as filtering, transformation, and color manipulation.
* **Computer Vision Algorithms:** The library includes implementation of various computer vision algorithms, including feature detection, object recognition, and image stitching.
* **Machine Learning:** OpenCV integrates with machine learning frameworks and provides tools for training and deploying machine learning models. This is particularly useful for tasks like object detection and facial recognition.
* **Camera Calibration**: OpenCV includes functions for camera calibration, essential in computer vision applications to correct distortions caused by camera lenses.
* **Real-time Computer Vision:** It supports real-time computer vision applications, making it suitable for tasks like video analysis, motion tracking, and augmented reality.
* **Cross-Platform Support:** OpenCV is compatible with various operating systems, including Windows, Linux, macOS, Android, and iOS. This makes it versatile for a wide range of applications.
* **Community Support**: With a large and active community, OpenCV is continuously evolving, with contributions from researchers, developers, and engineers worldwide.

OpenCV is widely used in academia, industry, and research for tasks ranging from simple image manipulation to complex computer vision and machine learning applications. Its versatility and comprehensive set of tools make it a go-to library for developers working in the field of computer vision.

**An Overview of imgbeddings**

Here’s a[Python package to generate embedding](https://github.com/minimaxir/imgbeddings) vectors from images, using [OpenAI](https://openai.com/)’s robust [CLIP model](https://github.com/openai/CLIP) via [Hugging Face](https://huggingface.co/) [transformers](https://huggingface.co/docs/transformers/index). These image embeddings, derived from an image model that has seen the entire internet up to mid-2020, can be used for many things: unsupervised clustering (e.g. via [umap](https://umap-learn.readthedocs.io/en/latest/)), embeddings search (e.g. via [faiss](https://github.com/facebookresearch/faiss)), and using downstream for other framework-agnostic ML/AI tasks such as building a classifier or calculating image similarity.

* The [embeddings generation models](https://huggingface.co/minimaxir/imgbeddings) are ONNX INT8-quantized — meaning they’re 20–30% faster on the CPU, much smaller on disk, and don’t require PyTorch or TensorFlow as a dependency!
* Works for many different image domains, thanks to CLIP’s zero-shot performance.
* Includes utilities for using [principal component analysis (PCA)](https://en.wikipedia.org/wiki/Principal_component_analysis) to reduce the dimensionality of generated embeddings without losing much info.

**Vector Store Explained**

**Definition**

Vector stores are specialized databases designed for efficient storage and retrieval of vector embeddings. This specialization is crucial, as conventional databases like SQL are not finely tuned for handling extensive vector data.

**Role of Embeddings**

Embeddings represent data, typically unstructured data like text or images, in numerical vector formats within a high-dimensional space. Traditional relational databases are ill-suited for storing and retrieving these vector representations.

**Key Features of Vector Stores**

* Efficient Indexing: Vector stores can index and rapidly search for similar vectors using similarity algorithms.
* Enhanced Retrieval: This functionality allows applications to identify related vectors based on a provided target vector query.

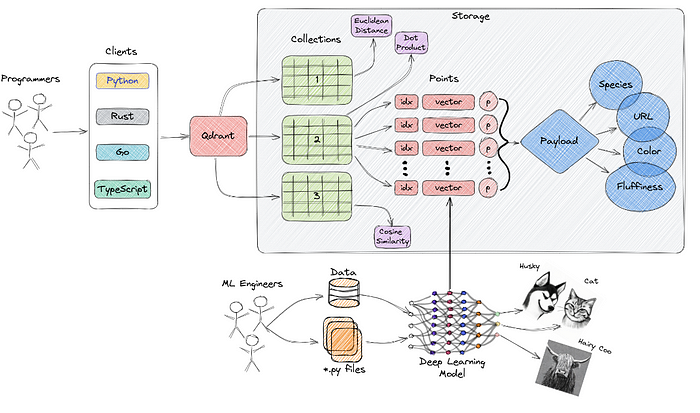
**An Overview of Qdrant**

<https://qdrant.tech/documentation>

Qdrant is a specialized vector similarity search engine designed to offer a production-ready service through a user-friendly API. It facilitates the storage, search, and management of points (vectors) along with additional payloads. These payloads serve as supplementary pieces of information, enhancing the precision of searches and providing valuable data to users.

Getting started with Qdrant is seamless. Utilize the Python qdrant-client, access the latest Docker image of Qdrant and establish a local connection, or explore Qdrant’s Cloud free tier option until you are prepared for a comprehensive transition.

**High-Level Qdrant Architecture**



**Understanding Semantic Similarity**

Semantic similarity, in the context of a set of documents or terms, is a metric that gauges the distance between items based on the similarity of their meaning or semantic content, rather than relying on lexicographical similarities. This involves employing mathematical tools to assess the strength of the semantic relationship between language units, concepts, or instances. The numerical description obtained through this process results from comparing the information that supports their meaning or describes their nature.

It’s crucial to distinguish between semantic similarity and semantic relatedness. Semantic relatedness encompasses any relation between two terms, whereas semantic similarity specifically involves ‘is a’ relation. This distinction clarifies the nuanced nature of semantic comparisons and their application in various linguistic and conceptual contexts.

**Code Implementation**

**Install required dependencies**

pip install qdrant-client imgbeddings pillow opencv-python

**Create a folder to store the required images**

mkdir photos

**Download the Model Parameter file**

Download the **haarcascade\_frontalface\_default.xml pre-trained**Haar Cascade model from the [OpenCV GitHub repository](https://github.com/opencv/opencv/tree/master/data/haarcascades) and store it locally.

This code is implemented using Python on Google Colab.

**Sample Code**

**Import required dependencies**

#import required libraries  
import cv2  
import numpy as np  
from imgbeddings import imgbeddings  
from PIL import Image

**Helper function to extract face from images**

def detect\_face(image\_path,target\_path):  
 # loading the haar case algorithm file into alg variable  
 alg = "haarcascade\_frontalface\_default.xml"  
 # passing the algorithm to OpenCV  
 haar\_cascade = cv2.CascadeClassifier(alg)  
 # loading the image path into file\_name variable  
 file\_name = image\_path  
 # reading the image  
 img = cv2.imread(file\_name, 0)  
 # creating a black and white version of the image  
 gray\_img = cv2.cvtColor(img, cv2.COLOR\_RGB2BGR)  
 # detecting the faces  
 faces = haar\_cascade.detectMultiScale(gray\_img, scaleFactor=1.05, minNeighbors=2, minSize=(100, 100))  
 # for each face detected  
 for x, y, w, h in faces:  
 # crop the image to select only the face  
 cropped\_image = img[y : y + h, x : x + w]  
 # loading the target image path into target\_file\_name variable  
 target\_file\_name = target\_path  
 cv2.imwrite(  
 target\_file\_name,  
 cropped\_image,  
 )

The below code is responsible

*faces = haar\_cascade.detectMultiScale(gray\_img, scaleFactor=1.05, minNeighbors=2, minSize=(100, 100))*

Where:

* gray\_img — the source image where we need to find faces.
* scaleFactor — the scaling factor; the higher ratio, the more compression and more loss in image quality.
* minNeighbors — the amount of neighbor faces to collect. The higher, the more of the same face could appear multiple times.
* minSize — the minimum size of a detected face, in this case a square of 100 pixels.

The for loop iterates over all the faces detected and stores them in separated files. You might want to define a variable (maybe using the x and y parameters) to store the various faces in different files.

The result of the face detection stage is not perfect: it identifies three faces out of the four that are visible, but is good enough for our purpose. You can fine-tune the algorithm parameters to find the better fit for your use cases.

**Helper function to calculate the embeddings**

def generate\_embeddings(image\_path):  
 #  
 # loading the face image path into file\_name variable  
 file\_name = "/content/target\_photo\_1.jpg"  
 # opening the image  
 img = Image.open(file\_name)  
 # loading the `imgbeddings`  
 ibed = imgbeddings()  
 # calculating the embeddings  
 embedding = ibed.to\_embeddings(img)[0]  
 emb\_array = np.array(embedding).reshape(1,-1)  
 return emb\_array

**Detect faces from images and convert into gray scale images in target folder**

os.mkdir("target")  
# loop through the images in the photos folder and extract faces  
file\_path = "/content/photos"  
for item in os.listdir(file\_path):  
 if item.endswith(".jpeg"):  
 detect\_face(os.path.join(file\_path,item),os.path.join("/content/target",item))

**Loop through faces extracted from target folder and generate embeddings**

img\_embeddings = [generate\_embeddings(os.path.join("/content/target",item)) for item in os.listdir("/content/target")]  
print(len(img\_embeddings))  
#  
print(img\_embeddings[0].shape)  
#  
#save the vector of embeddings as a NumPy array so that we don't have to run it again later  
np.save("vectors\_cv2", np.array(img\_embeddings), allow\_pickle=False)

**Set up Vector Store to store image embeddings**

# Create a local Qdrant vector store  
client =QdrantClient(path="qdrant\_db\_cv2")  
#  
my\_collection = "image\_collection\_cv2"  
client.recreate\_collection(  
 collection\_name=my\_collection,  
 vectors\_config=models.VectorParams(size=768, distance=models.Distance.COSINE)  
)  
# generate metadata  
payload = []  
files\_list= os.listdir("/content/target")  
for i in range(len(os.listdir("/content/target"))):  
 payload.append({"image\_id" :i,  
 "name":files\_list[i].split(".")[0]})  
  
print(payload[:3])  
ids = list(range(len(os.listdir("/content/target"))))  
#Load the embeddings from the save pickle file  
embeddings = np.load("vectors\_cv2.npy").tolist()  
#  
# Load the image embeddings  
for i in range(0, len(os.listdir("/content/target"))):  
 client.upsert(  
 collection\_name=my\_collection,  
 points=models.Batch(  
 ids=[ids[i]],  
 vectors=embeddings[i],  
 payloads=[payload[i]]  
 )  
 )

**Make sure vectors are uploaded successfully by counting them**

client.count(  
 collection\_name=my\_collection,  
 exact=True,  
)  
##Response  
CountResult(count=6)

**Visually inspect the collection created**

client.scroll(  
 collection\_name=my\_collection,  
 limit=10  
)

**Image Search**

Load the new image and extract the face

load\_image\_path = '/content/target/Aishw.jpeg'  
target\_image\_path = 'black.jpeg'  
detect\_face(load\_image\_path,target\_path)

**Check the saved image**

Image.open("/content/black.jpeg")



Gray Scale cropped face image saved

**Generate the image embedding**

query\_embedding = generate\_embeddings("/content/black.jpeg")  
print(type(query\_embedding))  
#  
print(query\_embedding.shape)  
  
##Response  
numpy.ndarray  
(1, 768)

**Search for images to recognize the supplied input image**

results = client.search(  
 collection\_name=my\_collection,  
 query\_vector=query\_embedding[0],  
 limit=5,  
 with\_payload=True  
)  
print(results)  
files\_list= [ os.path.join("/content/target",f) for f in os.listdir("/content/target")]  
print(files\_list)  
  
##Response  
  
[ScoredPoint(id=3, version=0, score=0.9999998807907104, payload={'image\_id': 3, 'name': 'Aishw'}, vector=None, shard\_key=None),  
 ScoredPoint(id=2, version=0, score=0.9999998807907104, payload={'image\_id': 2, 'name': 'deepika'}, vector=None, shard\_key=None),  
 ScoredPoint(id=1, version=0, score=0.9999998807907104, payload={'image\_id': 1, 'name': 'nohra'}, vector=None, shard\_key=None),  
 ScoredPoint(id=0, version=0, score=0.9999998807907104, payload={'image\_id': 0, 'name': 'kajol'}, vector=None, shard\_key=None),  
 ScoredPoint(id=5, version=0, score=0.9999998211860657, payload={'image\_id': 5, 'name': 'kareena'}, vector=None, shard\_key=None)]  
  
  
['/content/target/kajol.jpeg',  
 '/content/target/nohra.jpeg',  
 '/content/target/deepika.jpeg',  
 '/content/target/Aishw.jpeg',  
 '/content/target/aish.jpeg',  
 '/content/target/kareena.jpeg']

**Helper function to display the result**

def see\_images(results, top\_k=2):  
 for i in range(top\_k):  
 image\_id = results[i].payload['image\_id']  
 name = results[i].payload['name']  
 score = results[i].score  
 image = Image.open(files\_list[image\_id])  
  
 print(f"Result #{i+1}: {name} was diagnosed with {score \* 100} confidence")  
 print(f"This image score was {score}")  
 display(image)  
 print("-" \* 50)  
 print()

**Display Search Results- show top 5 matching images**

see\_images(results, top\_k=5)

**Results of the image Search**

Result #1: Aishw was diagnosed with 99.99998807907104 confidence.

This image score was 0.9999998807907104.



— — — — — — — — — — — — — — — — — — — — — — — — —

Result #2: deepika was diagnosed with 99.99998807907104 confidence.

This image score was 0.9999998807907104.



— — — — — — — — — — — — — — — — — — — — — — — — —

Result #3: nohra was diagnosed with 99.99998807907104 confidence.

This image score was 0.9999998807907104.



— — — — — — — — — — — — — — — — — — — — — — — — —

Result #4: kajol was diagnosed with 99.99998807907104 confidence.

This image score was 0.9999998807907104.



— — — — — — — — — — — — — — — — — — — — — — — — —

Result #5: kareena was diagnosed with 99.99998211860657 confidence.

This image score was 0.9999998211860657.



As you can see, we used an existing image and got back other images along with the original image. The similarity score also gives us a good indication regarding the similarity of our query image and those in our database.

**Method 2. Using Transformers and Qdrant for Image Recognition**

Apart from OpenCV, we can also use Vision Transformers to perform the same task. Please find the sample code below:

**Code implementation**

**Install required dependencies**

pip install -qU qdrant-client transformers datasets

**Import required libraries**

from transformers import ViTImageProcessor, ViTModel  
from qdrant\_client import QdrantClient  
from qdrant\_client.http import models  
from datasets import load\_dataset  
import numpy as np  
import torch

**Setup the vectorstore**

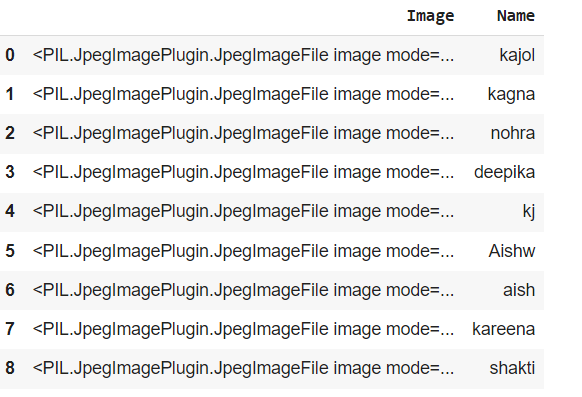
# Create a local Qdrant vector store  
client =QdrantClient(path="qdrant\_db")  
my\_collection = "image\_collection"  
client.recreate\_collection(  
 collection\_name=my\_collection,  
 vectors\_config=models.VectorParams(size=384, distance=models.Distance.COSINE)  
)

**Load the model**

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
processor = ViTImageProcessor.from\_pretrained('facebook/dino-vits16')  
model = ViTModel.from\_pretrained('facebook/dino-vits16').to(device)

**Preprocess the images in the photos folder and load into a dataframe**

import pandas as pd  
import os  
image\_file = []  
image\_name =[]  
#  
for file in os.listdir("/content/photos"):  
 if file.endswith(".jpeg"):  
 image\_name.append(file.split(".")[0])  
 image\_file.append(Image.open(os.path.join("/content/photos",file)))  
#  
df = pd.DataFrame({"Image":image\_file,"Name":image\_name})  
descriptions = df['Name'].tolist()  
print(descriptions)



Dataframe Snapshot

**Generate embeddings with ViTs**

In computer vision systems, vector databases are used to store image features. These image features are vector representations of images that capture their visual content, and they are used to improve the performance of computer vision tasks such as object detection, image classification, and image retrieval.

To extract these useful feature representations from our images, we’ll use vision transformers (ViT). ViTs are advanced algorithms that enable computers to “see” and understand visual information in a similar fashion to humans. They use a transformer architecture to process images and extract meaningful features from them.

To understand how ViTs work, imagine you have a large jigsaw puzzle with many different pieces. To solve the puzzle, you would typically look at the individual pieces, their shapes, and how they fit together to form the full picture. ViTs work in a similar way, meaning, instead of looking at the entire image at once, vision transformers break it down into smaller parts called “patches.” Each of these patches is like one piece of the puzzle that captures a specific portion of the image, and these pieces are then analyzed and processed by ViTs.

By analyzing these patches, ViTs identify important patterns such as edges, colors, and textures, and combine them to form a coherent understanding of a given image.

final\_embeddings = []  
for item in df['Image'].values.tolist():  
 inputs = processor(images=item, return\_tensors="pt").to(device)  
 with torch.no\_grad():  
 outputs = model(\*\*inputs).last\_hidden\_state.mean(dim=1).cpu().numpy()  
 final\_embeddings.append(outputs)

**Save the embeddings**

np.save("vectors", np.array(final\_embeddings), allow\_pickle=False)

**Generate the metadata**

payload = []  
for i in range(df.shape[0]):  
 payload.append({"image\_id" :i,  
 "name":df.iloc[i]['Name']})  
  
ids = list(range(df.shape[0]))  
embeddings = np.load("vectors.npy").tolist()

**Load the embeddings into a vector store**

for i in range(0, df.shape[0]):  
 client.upsert(  
 collection\_name=my\_collection,  
 points=models.Batch(  
 ids=[ids[i]],  
 vectors=embeddings[i],  
 payloads=[payload[i]]  
 )  
 )  
  
#check if the update is successful  
client.count(  
 collection\_name=my\_collection,  
 exact=True,  
)  
#To visually inspect the collection we just created, we can scroll through our vectors with the client.scroll() method.  
client.scroll(  
 collection\_name=my\_collection,  
 limit=10  
)

**Search for an image/photo from the data store**

img = Image.open("YOUR IMAGE PATH")  
inputs = processor(images=img, return\_tensors="pt").to(device)  
one\_embedding = model(\*\*inputs).last\_hidden\_state  
#  
results = client.search(  
 collection\_name=my\_collection,  
 query\_vector=one\_embedding.mean(dim=1)[0].tolist(),  
 limit=5,  
 with\_payload=True  
)  
see\_images(results, top\_k=2)

**Search Results**

**Original image to be searched**



**Search results**

Result #1: Aishw was diagnosed with 100.00000144622251 confidence.

The image score was 1.0000000144622252.



— — — — — — — — — — — — — — — — — — — — — — — — —

Result #2: deepika was diagnosed with 90.48531271076924 confidence.

The image score was 0.9048531271076924.

— — — — — — — — — — — — — — — — — — — — — — — — —

Result #3: nohra was diagnosed with 88.62201422801974 confidence.

The image score was 0.8862201422801974.



— — — — — — — — — — — — — — — — — — — — — — — — —

Result #4: aish was diagnosed with 87.71421890846095 confidence.

This image score was 0.8771421890846095.



— — — — — — — — — — — — — — — — — — — — — — — — —

Result #5: kareena was diagnosed with 86.80090570447916 confidence.

This image score was 0.8680090570447916.



— — — — — — — — — — — — — — — — — — — — — — — — —

**References**

* [https://qdrant.tech/documentation](https://qdrant.tech/documentation/)
* <https://github.com/opencv/opencv>

[connect with me](https://www.linkedin.com/in/plaban-nayak-a9433a25/)