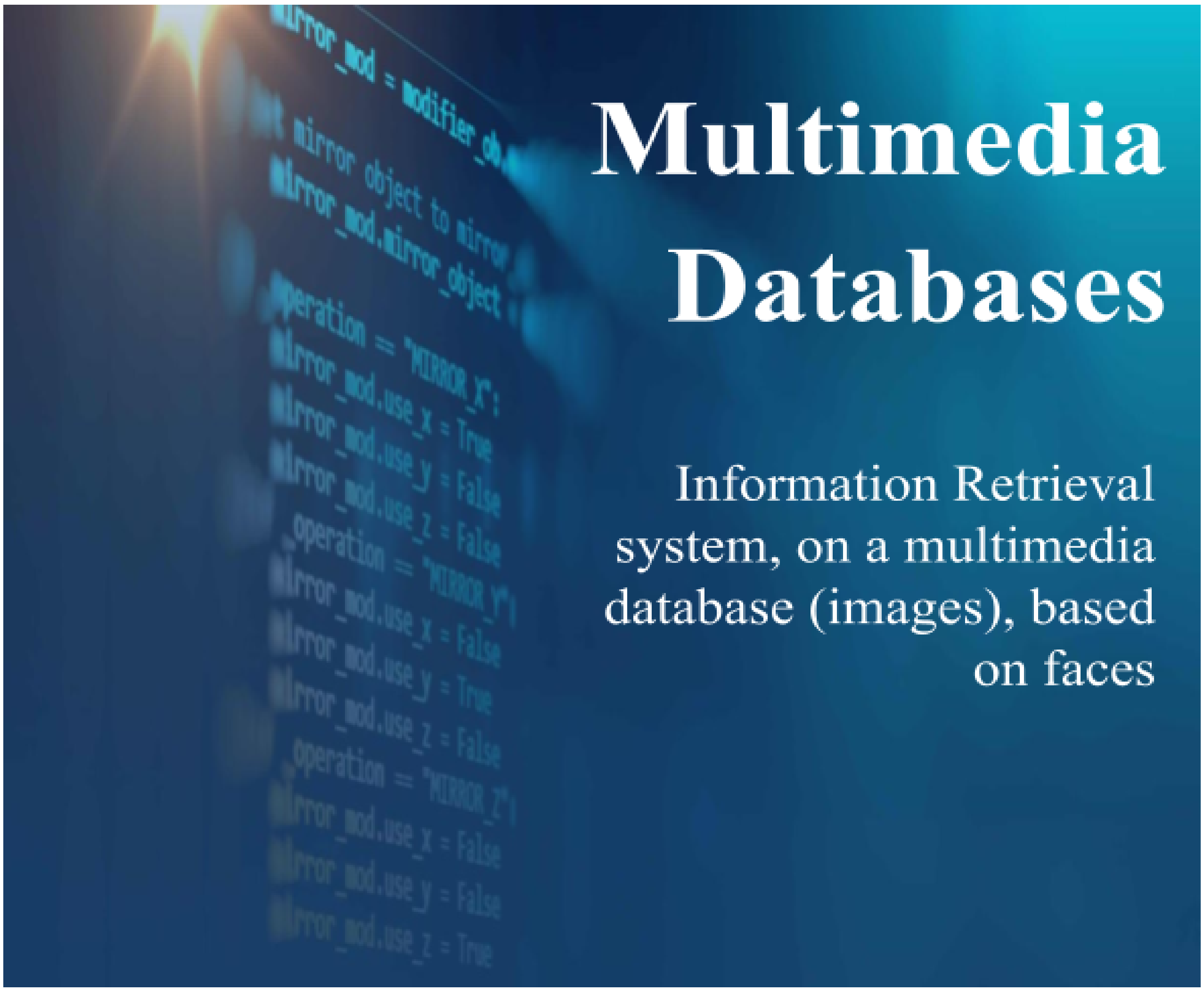
MSc Data and Web Science

Course: Advanced Topics in Databases, Spring Semester, 2020 - 2021

Professor: Eleftherios Tiakas



Multimedia Databases

Information Retrieval system, on a multimedia database (images), based on faces

Panagiotis Papaemmanouil
AEM: 56

Table of Contents

Introduction

Application Description

The scope of this project is the development of an Information Retrieval (IR) System for a Multimedia Database of images, based on the content of those images. More specifically, the system will give the ability to the user to search on the Images Database, based on the faces appearing in each photo.

For this purpose, we need to utilize ***face recognition*** algorithms along with ***pre-trained Deep Learning models***, for the identification of the faces in images and the numerical encoding of the identified faces (descriptor vectors extraction). Also, we use the ***PostgreSQL*** Database management system (DBMS), to store the face encoding vectors along with their metadata. For image gathering and preprocessing, as well as the implementation of the main application functionality (IR system) the ***Python*** programming language was used.

The full source code, the dataset, and every other material related to this project can be found in the **Github** repository: https://github.com/papaemman/multimedia_db

Application Architecture and Technology Stack

For this project, we decide to collect our dataset of images, so the first component of the Application is a web scraper. This ***web scraper*** is based on the *selenium* python library and it can automatically search and download multiple images from Google Images. After we collect enough images and store them on the disc, the next step is to preprocess them and store them in a PostgreSQL database.

The processing and storage steps are implemented in Python. At the first step, a face recognition module identifies all faces appearing in each photo, and then a face encoding module transforms these faces into numerical vectors. After that, these numerical vectors are stored in the PostgreSQL database along with metadata like the path of the original image, the name of the person appearing in the photo, etc. Hence, we create a Database consists of descriptor vectors for multiple faces and their respective metadata.

Finally, the user can submit queries in this Database in the form of images, along with a number k indicating the requested number of similar images to be returned. The input image will be then transformed into a descriptor vector using the face recognition and face encoding modules described above. This descriptor vector will be compared with every other descriptor vector stored in the database and the k most similar results will be returned as output. Here the term “similar” is referred to facial features similarity, so the output will consist of images having similar faces with the input image’s face.

Figure 1, presents a high-level architecture of the application along with its core components.

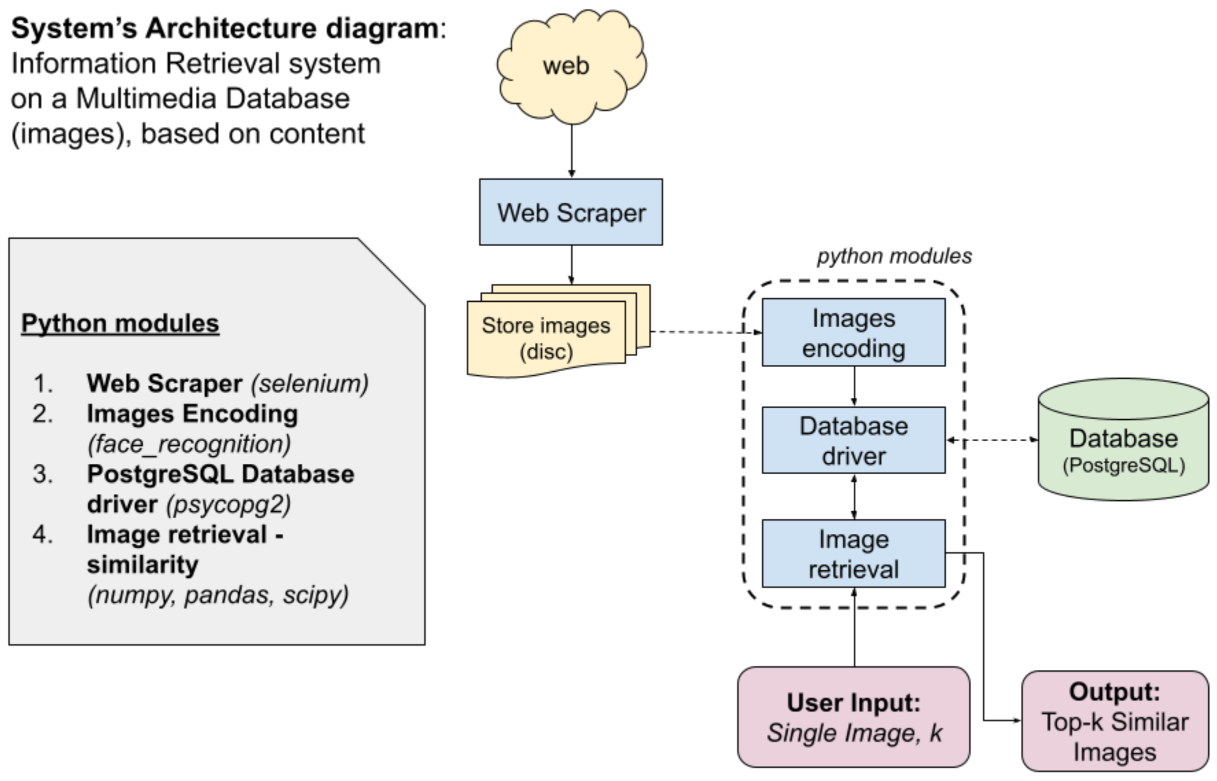

System's Architecture diagram:
Information Retrieval system
on a Multimedia Database
(images), based on content
web
Web Scraper
Store images (disc)
python modules
Images encoding
Database driver
Image retrieval
Database (PostgreSQL)
User Input: Single Image, k
Output: Top-k Similar Images
Python modules
1. Web Scraper (selenium)
2. Images Encoding (face_recognition)
3. PostgreSQL Database driver (psycopg2)
4. Image retrieval - similarity (numpy, pandas, scipy)

Figure 1: System's Architecture Diagram

As for the Technology Stack, we used **VSCode** and **Jupyter Notebooks,** for python development, while for PostgreSQL DBMS we use the **pgAdmin4** management tool.

Main Python Libraries

1. **Selenium**: Web scrapping functionalities
2. **psycopg2**: Python driver for PostgreSQL
3. **face_recognition**: Face recognition and Face encoding functionalities
4. **pandas**: Data preprocessing
5. **scipy**: Scientific calculations (distance measure)
6. **PIL**: Python Imaging Library

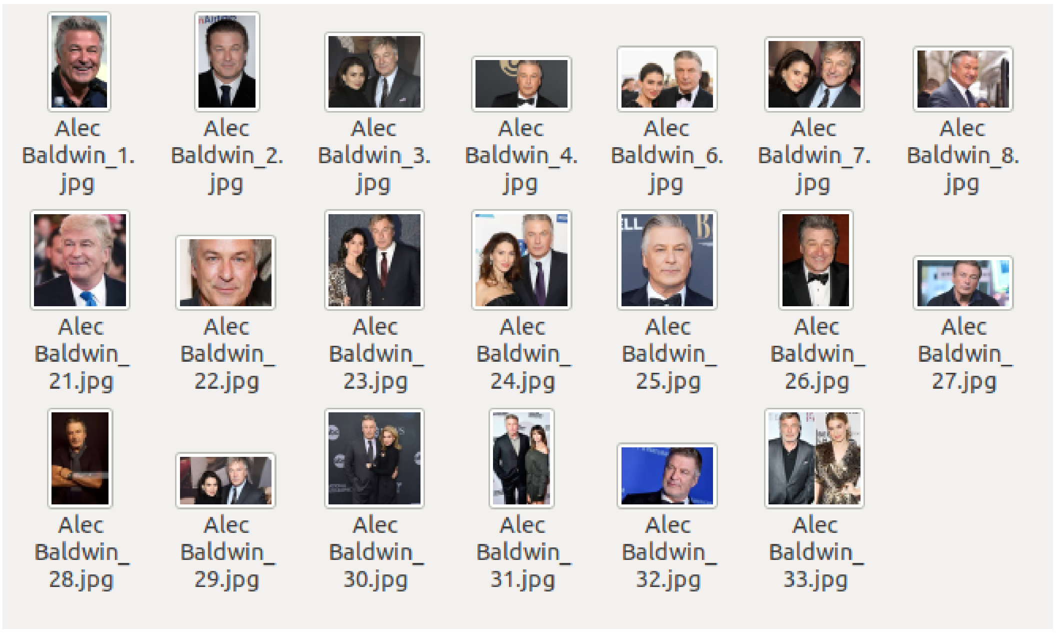
Dataset

As already mentioned in the previous section, the dataset we used for this project was collected from the web, using a ***web scrapping*** script written in python, based on the selenium library. We wrote a custom **web scraper,** which was able to get a *name* as an argument and then to search for this name in Google images and download the first N images appearing in the results. We decided to collect photos of famous Hollywood actors. So, we managed to collect more than **3.000 images**, for **87 Hollywood actors**, in less than 2 hours.

Below we provide the Full list of the actors :

```
Robert De Niro, Al Pacino, Morgan Freeman, Will SmithJack Nicholson, Harrison
Ford,GérardDepardieu, Dustin Hoffman, Tom Cruise, Heath Ledger, Anthony Hopkins,
Eddie Murphy,Antonio Banderas, Robin Williams, Samuel L. Jackson, Charles Chaplin,
Bruce Lee, Johnny Depp, Leonardo DiCaprio, Tom Hanks, Daniel Day-Lewis, Denzel
Washington, Marlon Brando, Tommy LeeJones, Robert Redford,Joaquin
Phoenix,Christopher Walken,George Clooney,Bruce Willis, SeanConnery,Russell
Crowe,Nicolas Cage,Brad Pitt,Kevin Costner,Clint Eastwood,Donald Sutherland,Michael
Douglas,Robert Downey Jr., Mel Gibson, Jim Carrey, Mark Wahlberg, Christoph Waltz,
Arnold Schwarzenegger, Sylvester Stallone, Alec Baldwin, Mads Mikkelsen, Ben
Stiller, Willem Dafoe, Ed Harris, Matt LeBlanc, Matthew Perry, David Schwimmer,
Jennifer Aniston, Courteney Cox, Lisa Kudrow, Whoopi Goldberg , Uma Thurman , Winona
Ryder , Drew Barrymore , Angelina Jolie , Sandra Bullock, Penélope Cruz , Nicole
Kidman , Lucy Liu, Julia Roberts, Halle Berry , Marilyn Monroe, Sophia Loren, Kate
Winslet, Audrey Hepburn, Cate Blanchett, Elizabeth Taylor, Meryl Streep, Amy Adams,
Hilary Swank, Diane Lane, Frances McDormand, Judi Dench, Sharon Stone, Kim Basinger,
Anne Hathaway, Demi Moore, Liza Minnelli, Barbra Streisand, Jennifer Lawrence,
Gwyneth Paltrow, Charlize Theron.
```

For every actor, all their photos were stored in a single directory, so they not get confused with other actors' faces. Below, we provide a screenshot of Alec Baldwin's directory.



About images: After an initial investigation of the downloaded images, we can conclude that the web scrapper downloads various images with different characteristics. For example, there are images with and without ***color***. Images in which only a ***single face*** appears and others with ***many*** faces in them, various ***face angles*** (anfas + profile), etc. Therefore, we need to make sure that the face recognition and the face encoding modules can work accurately in all of these cases.

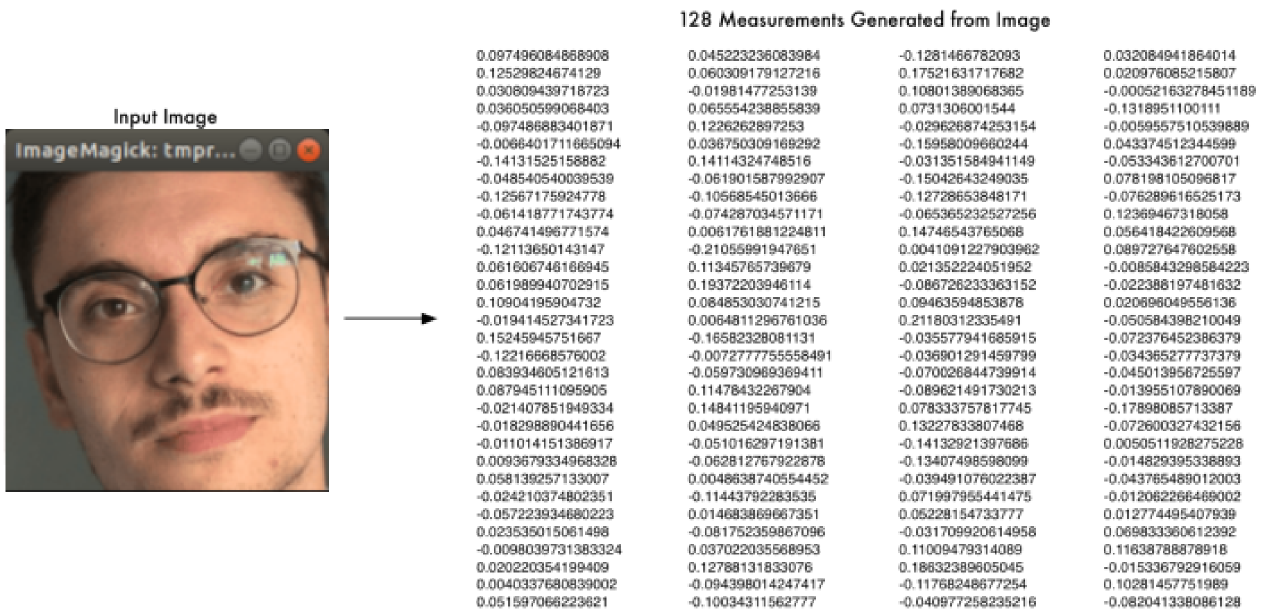
Database Storage (PostgreSQL)

In the previous section, we present the data collection process along with a brief description of the collected dataset. In this section, we will move forward to describe in-depth the process used to extract descriptor vectors from every image's faces and how we can import them in PostgreSQL.

Descriptor vectors: Face Recognition and Face Encoding

For our application to support searching based on similar faces, we need a **face recognition** and **face encoding** functionality. For this purpose, we use the ***face_recognition library*** (ageitgey/face_recognition: The world's simplest facial recognition API for Python and the command line). This library offers a variety of functions for many face recognition-related tasks such as: find faces in pictures, find landmarks of a face, crop faces, find and manipulate facial features, identify faces in pictures, etc. Thankfully, this library can work with many different types of photos and therefore can tackle the problems described in the previous section, about the different types of collected photos.

Face encoding serves as a tool to quantify this similarity between 2 different faces. Fortunately, this problem has been solved using the Deep Learning technique, and therefore there are plenty of out-of-the-self pre-trained models available. These Deep Learning models are based on Convolutional layers and are trained to generate a constant number of measurements based on a face photo. **Figure 2** demonstrates visually an example of the mapping of an Face image into a 128-d numerical vector.



0.097496084868908	0.045223236083984	-0.1281466782093	0.032084941864014
0.12529824674129	0.060309179127216	0.17521631717682	0.020976085215807
0.030809439718723	-0.01981477253139	0.10801389068365	-0.00052163278451189
0.036050599068403	0.065554238855839	0.0731306001544	-0.1318951100111
-0.097486883401871	0.1226262897253	-0.029626874253154	-0.0059557510539889
-0.0066401711665094	0.036750309169292	-0.15958009660244	0.043374512344599
-0.14131525158882	0.14114324748516	-0.031351584941149	-0.053343612700701
-0.048540540039539	-0.061901587992907	-0.15042643249035	0.078198105096817
-0.12567175924778	-0.10568545013666	-0.12728653848171	-0.076289616525173
-0.061418771743774	-0.074287034571171	-0.065365232527256	0.12369467318058
0.046741496771574	0.0061761881224811	0.14746543765068	0.056418422609568
-0.12113650143147	-0.21055991947651	0.0041091227903962	0.089727647602558
0.061606746166945	0.11345765739679	0.021352224051952	-0.0085843298584223
0.061989940702915	0.19372203946114	-0.086726233363152	-0.022388197481632
0.10904195904732	0.084853030741215	0.09463594853878	0.020696049556136
-0.019414527341723	0.0064811296761036	0.21180312335491	-0.050584398210049
0.15245945751667	-0.16582328081131	-0.035577941685915	-0.072376452386379
-0.12216668576002	-0.0072777755558491	-0.036901291459799	-0.034365277737379
0.083934605121613	-0.059730969369411	-0.070026844739914	-0.045013956725597
0.087945111095905	0.11478432267904	-0.089621491730213	-0.013955107890069
-0.021407851949334	0.14841195940971	0.078333757817745	-0.17898085713387
-0.018298890441656	0.049525424838066	0.13227833807468	-0.072600327432156
-0.011014151386917	-0.051016297191381	-0.14132921397686	0.0050511928275228
0.0093679334968328	-0.062812767922878	-0.13407498598099	-0.014829395338893
0.058139257133007	0.0048638740554452	-0.039491076022387	-0.043765489012003
-0.024210374802351	-0.11443792283535	0.071997955441475	-0.012062266469002
-0.057223934680223	0.014683869667351	0.05228154733777	0.012774495407939
0.023535015061498	-0.081752359867096	-0.031709920614958	0.069833360612392
-0.0098039731383324	0.037022035568953	0.11009479314089	0.11638788878918
0.020220354199409	0.12788131833076	0.18632389605045	-0.015336792916059
0.0040337680839002	-0.094398014247417	-0.11768248677254	0.10281457751989
0.051597066223621	-0.10034311562777	-0.040977258235216	-0.082041338086128

Figure 2: Face encoding. A **128-d** numerical vector is generated based on the input image.

Working with PostgreSQL and Python

After we create the descriptor vector for each face of an image, we need to store it in PostgreSQL. To do this we utilize the ***psycopg2*** python library**,** which is a python driver for PostgreSQL. Using this library we can interact with the PostgreSQL database, from within the python environment and perform any action we want such as create, update and drop tables, submit SQL queries to filter data, etc. Furthermore, we can manually interact with PostgreSQL using the **pgAdmin 4** management tool to perform the same actions.

We can use the ***psycopg2*** library to connect to a running PostgreSQL database using the following snippet:

```
# Connect to DB
con = psycopg2.connect(
   host = "localhost",
   database = "images",
   user = "postgres",
   password = "pass",
   port = 5432
)

# Define the Cursor
cur = con.cursor()
```

The first step we need to do working with the PostgreSQL database is to create a new Database called ***images*** and then to create a new table in this database called ***actors***. This table will be used to store the descriptor vectors along with some other useful metadata such as the path for the original image, the actor's name, etc. This table will have a serial column called ***id*** as the Primary key. The code to define this table is presented below.

```
# Create new Table in images database
cur.execute(
   """
   CREATE TABLE actors (
       id SERIAL PRIMARY KEY,
       face_encoding BYTEA,
       image_path VARCHAR,
       actor VARCHAR
   )
   """)
```

After defining the ***actors*** table, it's time to iterate over all faces and store their respective values. These values are:

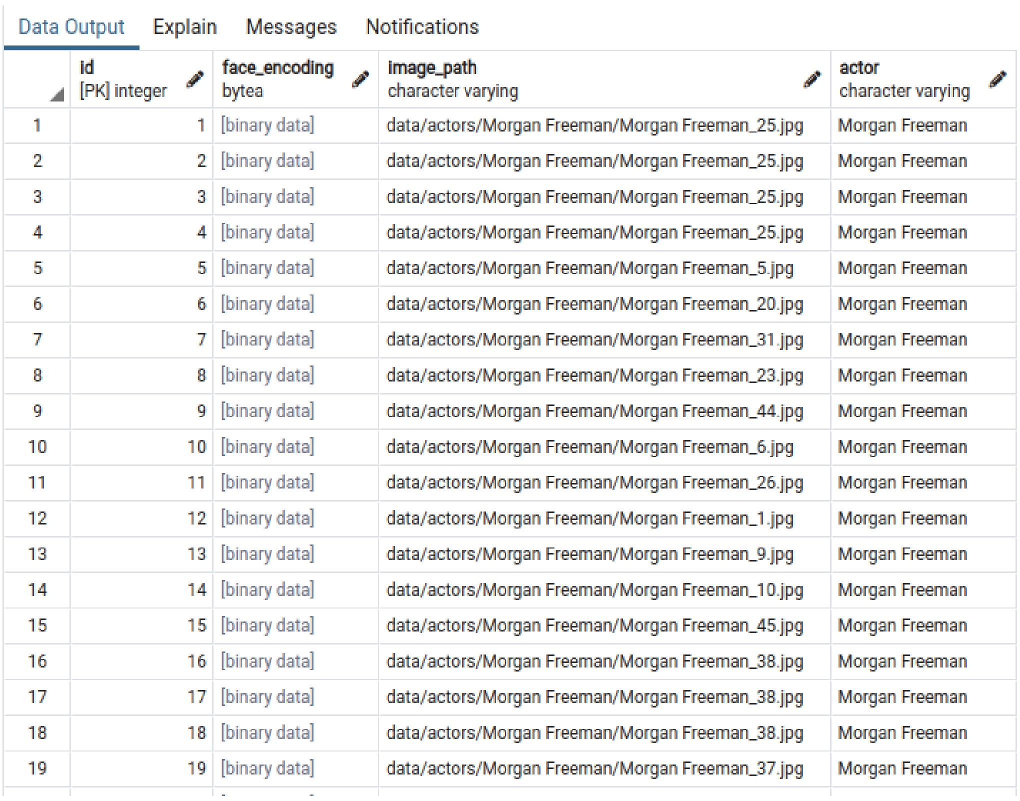
1. **the descriptor vector** (face_encoding) as BYTEA type
2. **the path for the original image** (image_path) as VARCHAR type, and
3. **the name of the actor** (actor) into the database.

An important note is that we don’t need to explicitly define the ***id*** (Primary key) of each entry because this will automatically be filled by PostgreSQL. The following snippet of code demonstrated this process.

```
cur.execute(
           """
           INSERT INTO actors(face_encoding, image_path, actor)
           VALUES (%s, %s,  %s)
           """,
           (pickle.dumps(faces_info[i]['face_encoding']),
faces_info[i]['image_path'], faces_info[i]['actor'])
       )
```

Finally, after we do the data entry, we need to specifically commit the transaction to the database. This will update the database and we will be able to see the new data added from python into the database (**Figure 3)**. After that, we can safely close the cursor and database connection.

```
# Commit the transaction
con.commit()
# Close the cursor
cur.close()
# Close the connection
con.close()
```



	id [PK] integer	face_encoding bytea	image_path character varying	actor character varying
1	1	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_25.jpg	Morgan Freeman
2	2	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_25.jpg	Morgan Freeman
3	3	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_25.jpg	Morgan Freeman
4	4	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_25.jpg	Morgan Freeman
5	5	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_5.jpg	Morgan Freeman
6	6	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_20.jpg	Morgan Freeman
7	7	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_31.jpg	Morgan Freeman
8	8	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_23.jpg	Morgan Freeman
9	9	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_44.jpg	Morgan Freeman
10	10	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_6.jpg	Morgan Freeman
11	11	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_26.jpg	Morgan Freeman
12	12	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_1.jpg	Morgan Freeman
13	13	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_9.jpg	Morgan Freeman
14	14	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_10.jpg	Morgan Freeman
15	15	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_45.jpg	Morgan Freeman
16	16	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_38.jpg	Morgan Freeman
17	17	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_38.jpg	Morgan Freeman
18	18	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_38.jpg	Morgan Freeman
19	19	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_37.jpg	Morgan Freeman

Figure 3: The ***“actors”*** table as can be seen in the **pgAdmin4** management tool (screenshot).

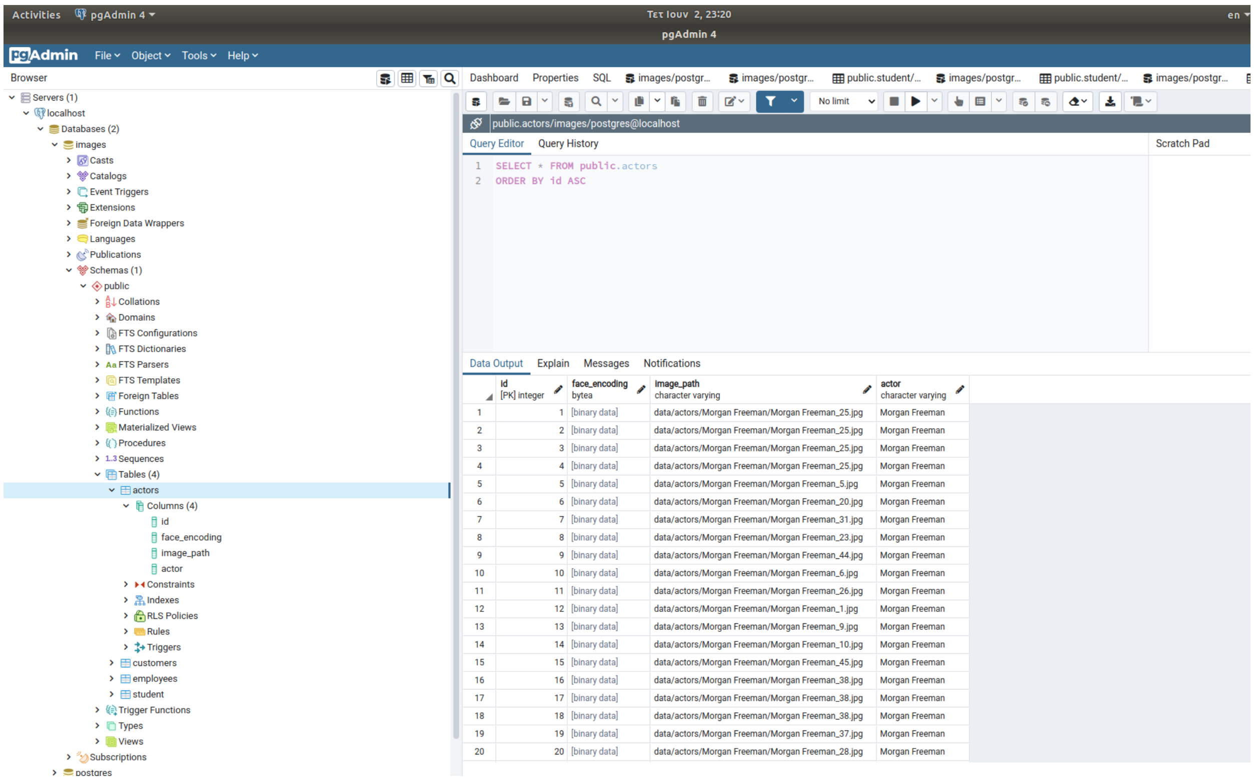
pgAdmin4 and SQL

Before we move on to the implementation of the Information Retrieval system, we run a series of **SQL queries** on the ***images*** database and more specifically in the ***actors*** table to examine its exact properties. For these queries, we use the ***Query Tool*** and the ***Query Editor***, provided by the pgAdmin4 management tool *(Figure 4)*.

About SQL: SQL stands for Structured Query Language and it is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems.

```
# Check all rows from actors table in images DB
SELECT *
FROM public.actors;

# Count the total number of rows for the actors table
SELECT COUNT(*)
FROM public.actors;

# Count the total actors
SELECT COUNT(DISTINCT actor)
FROM public.actors;

# Check all unique actors in DB
SELECT DISTINCT actor
FROM public.actors;

# Count total faces per photo (image_path)
SELECT image_path, COUNT(*) as total_faces
FROM public.actors
GROUP BY image_path
ORDER BY total_faces DESC;

# Filter rows for a specific image_path
SELECT *
FROM public.actors
WHERE image_path LIKE 'data/actors/Anne Hathaway/Anne Hathaway_46.jpg'
```

More screenshots and examples of these queries can be found here:
https://github.com/papaemman/multimedia_db/tree/main/output/sql-queries-pgAdmin4-screenshots



```sql
SELECT * FROM public.actors
ORDER BY id ASC
```

	id [PK] integer	face_encoding bytea	image_path character varying	actor character varying
1	1	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_25.jpg	Morgan Freeman
2	2	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_25.jpg	Morgan Freeman
3	3	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_25.jpg	Morgan Freeman
4	4	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_25.jpg	Morgan Freeman
5	5	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_5.jpg	Morgan Freeman
6	6	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_20.jpg	Morgan Freeman
7	7	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_31.jpg	Morgan Freeman
8	8	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_23.jpg	Morgan Freeman
9	9	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_44.jpg	Morgan Freeman
10	10	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_6.jpg	Morgan Freeman
11	11	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_26.jpg	Morgan Freeman
12	12	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_1.jpg	Morgan Freeman
13	13	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_9.jpg	Morgan Freeman
14	14	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_10.jpg	Morgan Freeman
15	15	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_45.jpg	Morgan Freeman
16	16	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_38.jpg	Morgan Freeman
17	17	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_38.jpg	Morgan Freeman
18	18	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_38.jpg	Morgan Freeman
19	19	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_37.jpg	Morgan Freeman
20	20	[binary data]	data/actors/Morgan Freeman/Morgan Freeman_28.jpg	Morgan Freeman

Figure 4: SQL query to print all rows of ***"actors"*** table (pgAdmin4 screenshot)

Information Retrieval System

The final component of this application is the implementation of an Information Retrieval system, on top of the PostgreSQL database providing the capability to search based on similar faces. More specifically, the user will be able to submit a single image with a single face as a search query along with an integer k and the system will perform a **k-NN (or k-Nearest Neighbors) query** in the Database. The result of this query will be a collage of the top-k most similar faces (their images) along with their respective paths in the filesystem. This process runs in the python module ***Image retrieval.***

Distance measures

We examine 5 different distance measures, to compute the similarity between descriptor vectors. More specifically, we use the implementations of these measures based on the ***scipy*** python library (Distance computations (scipy.spatial.distance) — SciPy v1.6.3 Reference Guide).

```
from scipy.spatial.distance import minkowski, euclidean, cityblock, cosine,
mahalanobis
```

Below, we provide the full list of the distance measures used, along with their mathematical formulation.

1. **Minkowski distance:** $(u, v) \; = \left(\sum_{i=1}^{n}(u_i - v_i)^p\right)^{\frac{1}{p}}$

Special cases of Minkowski distance are the:

2. **Euclidean distance** (for p=2)

3. **Manhattan** or **City-block** distance (for p=1).

4. **Cosine distance**: $(u, v) \; = \; 1 \; - \; \frac{u \bullet v}{|u| \, |v|}$, where $|u| = \left(\sum_{i=1}^{n} u_i^2 \right)^{1/2}$

The nominator is the dot product and the denominator is the product of the magnitude of each vector.

5. **Mahalanobis distance**: $(u, v) \; = \; (u \; - \; v) \, V^{-1} (u \; - \; v)^T$, where V^{-1} is the inverse covariance matrix

Results of k-NN queries

This final section provides results for multiple k-Nearest-Neighbors (k-NN) queries for multiple values of k and different input images. We run queries for 3 different use cases:

1. Faces already stored in DB
2. Faces not stored in DB, but the other photos of the same person were stored in DB
3. Faces of new persons, unknown to DB

About Distance metrics

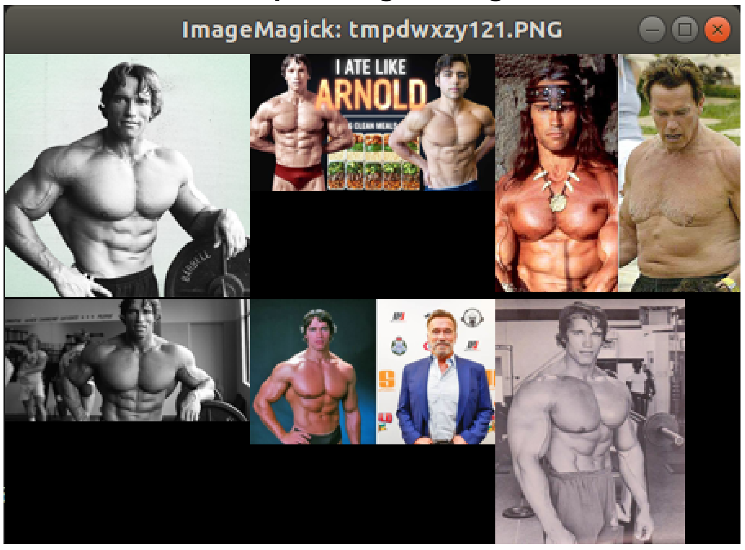
We run an extensive experimentation process, to evaluate the 5 distance measures presented in the previous subsection. This process consisted of multiple k-NN queries of the ***first and second use cases*** *(to have ground truth values)****,*** setting the number of k equal to the total number of faces stored in the Database for this actor. Based on these runs we conclude that the optimal distance metric was the ***euclidean distance***, with an outstanding precision of 99%. This means that for a k-NN query, 99% of the k returned images will correspond to the same face (person) as the input image.

Case 1: Faces already stored in DB

For this case, we run a single k-NN query, with k=6, using as input an image of Arnold Schwarzenegger already stored in DB. As we can see below, the IR system corrected identify the exact image as the most similar image. Also, all other images were images of Arnold Schwarzenegger, so this query has a precision of 100%.

Input image

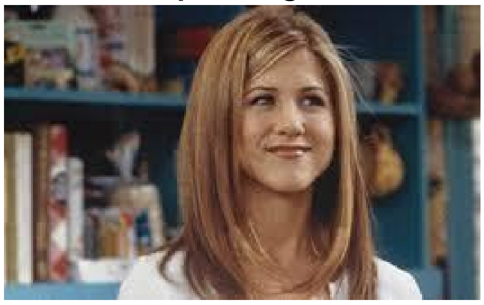
Output image collage



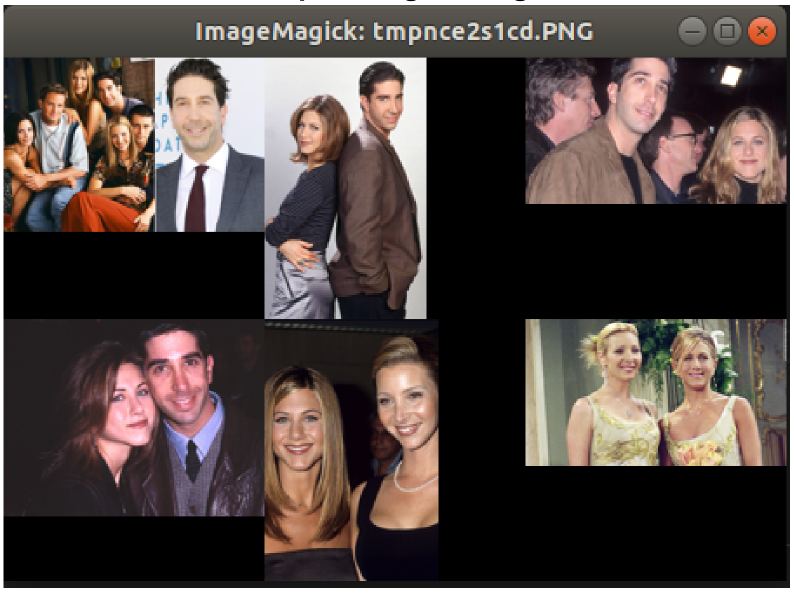
Case 2: Faces not stored in DB, but other photos of the same person are stored in DB.

For this case, we run a single k-NN query, with k=6, using as input a new image of Jennifer Aniston, different from all other images of her stored in DB. As we can see below, the IR system corrected identify 6 images in which the Jeniffer Aniston appear, so this query has a precision of 100%. We can see that in every image, there are other persons except for JA too. The IR system was able to return these photos because it automatically detects and encodes all faces appearing on each photo separately.

Input image



Output image collage

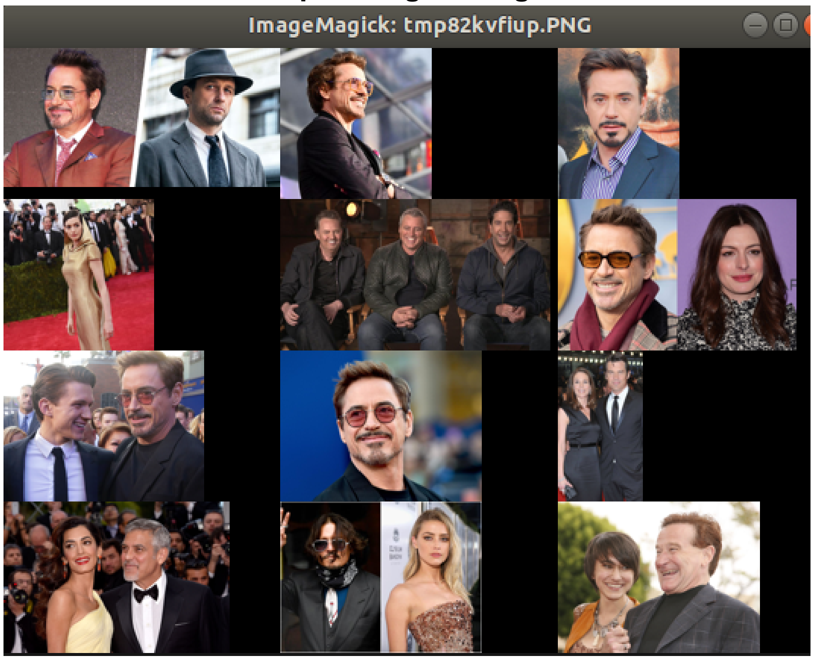


Case 3: Faces of new persons, unknown to DB

For this case, we run a single k-NN query, with k=12, using as input an image of a person unknown in the DB (me!). Looking at the results below, we can derive some interesting conclusions. First of all, from the 12 photos, the 6 are photos of Robert Downey Jr. This actor has a total of 43 images in the database, however only the 6 were returned here. Therefore we can support that the facial features between this actor and me are similar only on specific angles and facial expressions and not always. Except for Robert Downey Jr., there are 2 photos with many faces in the background. In these cases, the high similarity score appears with one of those faces. Finally, there are 4 more photos, all of them having different actors. We can't be sure about the exact face with a high similarity score with my face, without further investigation, so we categorize them as False Positives.

Input image

Output image collage


ImageMagick: tmp82kvfiup.PNG