UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Trabajo de Curso
Clasificador de personas mediante caras

Visión por Computador

Isac Añor Santana

Nikhil Chandru Durgadas Chellaram

Raúl Mateus Sánchez

Índice

Introducción

Este trabajo de curso consistirá en un clasificador de personas a partir de su cara. Esta propuesta tiene su motivación en herramientas como Google Photos, la cual cada cierto tiempo, agrupa en carpetas las fotos de las personas más recurrentes en la galería, de forma que, si necesitamos ver las fotos de alguien en particular, con acceder a su carpeta podamos tener un recopilatorio de las mismas.

Ante estas herramientas y su gran utilidad, nuestro grupo ha querido profundizar en ello, realizando una propuesta propia a partir de las técnicas y herramientas aprendidas durante la asignatura. Por ello, basado en DeepFace, presentaremos una propuesta de aplicación, con interfaz gráfica, que permita, a partir de un directorio o carpeta con imágenes, generar una clasificación de personas en base a sus caras en un directorio de salida mediante subcarpetas.

El código fuente, el desarrollo de esta propuesta así como las imágenes empleadas como conjunto (test-images) se puede encontrar en su correspondiente repositorio en GitHub: https://github.com/Isac-AS/40982-Trabajo-de-curso-VC/

Objetivos

Esta propuesta tiene como objetivo principal generar una clasificación de personas individuales mediante sus caras a partir de un conjunto de imágenes. Además, otro de los objetivos es hacer esta aplicación flexible y escalable para futuras ampliaciones o correcciones.

Por último, otro objetivo fundamental es ampliar el aprendizaje en el campo de la visión por computador, mediante la investigación realizada para esta propuesta, así como el estudio de resultados y el uso de herramientas ampliamente reconocidas.

Descripción técnica del desarrollo

Estructura e interfaz gráfica

La estructura e interfaz gráfica hace referencia a los aspectos del programa no relacionados directamente con la detección facial. El programa se ha construido de tal forma que su uso resulte fácil, y funciona de la siguiente manera:

Al ejecutar el programa, se inicializa la interfaz gráfica, que tiene dos acciones, la selección del directorio con las imágenes para procesar (inicialmente vacío en la primera ejecución) y el botón de acción que da lugar al procesamiento de las imágenes en el directorio seleccionado.

Cuando se selecciona un directorio, la ruta a éste se almacena en un fichero de preferencias con la finalidad de recordarlo para futuras ejecuciones.

Una vez se ha seleccionado el directorio con el álbum a procesar se puede hacer uso del botón de acción. Al pulsar el botón, se procede a crear un subdirectorio bajo "sorted_album", que se corresponderá con el directorio de salida del procesamiento. El procesamiento consiste en la identificación de las distintas caras en el álbum seleccionado y la creación de un subdirectorio bajo el directorio de salida por cada cara identificada y al finalizar el procesamiento, el resultado será que cada uno de estos subdirectorios contendrá las imágenes con caras de la misma persona.

La interfaz gráfica ha sido creada con la librería de Python *PyQt5* y la gestión de directorios con herramientas y paquetes nativos como *sys* y *os.*

Desarrollo del clasificador de caras

Una vez se cuenta con la estructura de la aplicación, se procede a desarrollar el "pipeline", el cual será el código que permitirá realizar la clasificación de caras a partir de un directorio de entrada que deberá contener imágenes. Para este desarrollo, principalmente se emplea DeepFace[1] y sus utilidades.

En el momento que el usuario haga click en *"Compute sort",* comenzará a tratar las imágenes ubicadas en la ruta que el usuario haya introducido para clasificarlas. Para ello, se llamará a la función compute_sort(), ubicada en el fichero pipeline.py.

Parte de este trabajo es investigar posibles formas de hacer una determinada tarea. En este caso, se han desarrollado tres formas de lograr un clasificador de caras. En primer lugar, se hace uso de los embeddings vistos en la práctica 6 de la asignatura, dentro del notebook Demo_BuscaParecidos, para hacer una primera prueba de la aplicación tomando embeddings de cada cara distinta que se detecte como referencia, es decir, si detecta 10 caras distintas, existirán 10 embeddings distintos para comparar. Aunque el desarrollo resultó satisfactorio, los resultados prácticamente eran insignificantes, pues se apreciaba demasiada confusión a pesar de la velocidad con la que procesaba las imágenes, pero sin precisión, de esta forma la herramienta era, a muchos efectos, inútil.

En segundo lugar, se pasa a emplear la herramienta DeepFace para el reconocimiento y detección de caras. Para ello, la estrategia a seguir es clara: se comparará cada imagen a clasificar con todas las imágenes ya clasificadas, en busca de la cara que más se parezca a la imagen a clasificar y así saber dónde debe ir esta imagen.

```
def compute_sort():
    """
    The sort action refers to create a directory for each of the faces
recognized
    within the images under the provided directory and fill each of the
created
    directories with the images that contain the respective face.
    """

    models = ["VGG-
Face","Facenet","Facenet512","OpenFace","DeepFace","DeepID","ArcFace","Dlib","
SFace"]
    metrics = ["cosine", "euclidean", "euclidean_l2"]
    backends = ['opencv','ssd','dlib','mtcnn','retinaface','mediapipe']

    input_dir = PreferenceFileHandler.get_base_directory()
    output_dir = PreferenceFileHandler.get_output_directory()

    image_file_extensions = [".jpeg", ".jpg", ".png"]
```

[1] https://github.com/serengil/deepface

```
    # Get a list of tuples with the path and names of the images in the input
directory
    with os.scandir(input_dir) as entries:
        images = [(entry.path, entry.name) for entry in entries if
os.path.splitext(entry.path)[1] in image_file_extensions]

    # Non elegant name of producing dir names
    faces_discovered_counter = 0
```

Común a todas las versiones de la aplicación, el pipeline contará con varios tipos de modelos, métricas y "*backends*" para hacer más fácil cualquier modificación a la hora de usar DeepFace. Por otra parte, gracias al fichero preferences.json y la clase PreferenceFileHandler, es posible obtener la ruta de salida donde estarán las imágenes clasificasas, así como la ruta con las imágenes que queremos clasificar.

Con esta última ruta, se obtiene la ruta y el nombre de cada una de las imágenes que se quiere clasificar. Las imágenes serán clasificadas en carpetas, cuyo nombre seguirá el patrón "face_X", siendo X un número que corresponderá a la variable faces_discovered_counter, la cual cuenta cuantas caras nuevas han sido descubiertas.

```
def face_detection(path, output_dir, model, metric):

    if (len(os.listdir(output_dir)) == 0):
        return 0, 0, 0

    aux_output_dir = output_dir + '/'
    df = DeepFace.find(img_path = path, db_path = aux_output_dir,
model_name=model, distance_metric = metric, prog_bar = False,
enforce_detection=False, silent = True)
    representations_path = f"{output_dir}/representations_arcface.pkl"
    os.remove(representations_path)

    if df.empty:
        return 0, 0, 0
    else:
        print(df.to_string())
        return 1, df.at[0, 'identity'], df.at[0, 'ArcFace_euclidean']
```

Para esta versión, se cuenta con una función llamada face_detection(), la cual recibe como parámetros la ruta a la imagen que se quiere clasificar, la ruta de la carpeta donde están las imágenes ya clasificadas y el modelo y métrica que se quiere usar. Se hace una rápida comprobación para ver si la imagen es la primera que se debe clasificar, puesto que en este caso, el directorio de salida estará vacío. Si es así, se devuelve 0, 0, 0, correspondiendo a status, path y distance. Status puede tomar valor 0 o 1, dependiendo de si es una cara nueva (0) o (1), y path y distance solo serán distinto de cero cuando se encuentre una cara parecida a la imagen pasada como parámetro, correspondiendo a la ruta

de la imagen que más se parece (para poder saber en que carpeta está y donde clasificarse) y la distancia.

Si el directorio de salida no está vacío, mediante DeepFace y su utilidad find se buscará parecidos de la imagen a clasificar en el directorio de salida. Si encuentra alguna imagen parecida, se entenderá que es la misma persona (puesto que es la finalidad de esta utilidad), y se devolverá como status 1, así como la ruta y distancia de la imagen más parecida. En caso contrario, se devolverá 0, 0, 0, al entender que es una nueva cara.

```
for image_path, image_name in images:
        # TODO
        # Face detection
        status, path, distance = face_detection(image_path, output_dir,
models[6], metrics[1])
        if status == 0:
            new_path = f"{output_dir}/face_{faces_discovered_counter}"
            faces_discovered_counter += 1

            # Directory creation
            os.mkdir(new_path)

            # Moving the file to the new directory
            output_path = f"{new_path}/{image_name}"
            # Copy the file
            shutil.copyfile(image_path, output_path)

        else:
            output_path = f"{os.path.dirname(path)}/{image_name}"
            shutil.copyfile(image_path, output_path)
```

Para cada imagen, se llamará a la función face_detection() antes explicada. Si el status obtenido es 0, se entiende como una nueva cara, y se crea la ruta donde irá esta imagen y se incrementará el contador de caras detectadas. Por último, se crea la carpeta que contendrá las imágenes de esta cara y allí se copiará esta imagen.

En caso de que el status fuese 1, se obtiene el directorio de la ruta de la imagen que más parecido tiene con la imagen a clasificar, y se forma la ruta añadiéndole el nombre de esta imagen, restando copiar allí el archivo.

Esta versión resulta en una mejoría exponencial la precisión al clasificar las imágenes por caras, pero con el inconveniente de que el tiempo de ejecución de la mayoría de los modelos y métricas probadas es demasiado elevado, y si se explicita *backends* como RetinaFace, la cosa empeora drásticamente no solo en tiempo, sino en funcionalidad, arrojando peores resultados. Esta versión se analizará con más detalle posteriormente.

Por último, nuestro objetivo queda enfocado en mantener o incluso mejorar los resultados obtenidos en la segunda versión, pero mejorando la velocidad de ejecución. Para

ello, se combinan las ideas de las dos versiones anteriores. Esto consiste en, mediante DeepFace, en vez de comparar cada imagen a clasificar con todas las imágenes ya clasificadas, se guardarán las nuevas caras detectadas para comparar con este subconjunto. En otras palabras, si nuestro directorio cuenta con 50 imágenes de 10 personas distintas, lo que ocurriría en un caso perfecto es que la última imagen a clasificar sea comparada con 10 imágenes de caras distintas, en vez de con 49. Por cada cara nueva detectada, su imagen será guardada en una carpeta auxiliar y las imágenes a clasificar se compararán con las caras ya detectadas mediante una imagen de referencia ubicadas aquí.

En cuánto al código, la parte común sigue siendo igual y no procede volver a detallarla. Por otro lado, la función face_detection() sufre un ligero cambio, pues antes se eliminaba después de cada detección el fichero “representations_MODELNAME.pkl”. Esto se realiza ya que, al agregar una nueva cara, para que la siguiente pueda ser comparada con esta, DeepFace debe volver a realizar la representación. En este caso, se elimina esta parte de la función, siendo reubicada.

```
def face_detection(path, output_dir, model, metric):

    if (len(os.listdir(output_dir)) == 0):
        return 0, 0, 0

    aux_output_dir = output_dir + '/'
    df = DeepFace.find(img_path = path, db_path = aux_output_dir,
model_name=model, distance_metric = metric, prog_bar = False,
enforce_detection=False, silent = True)

    if df.empty:
        return 0, 0, 0
    else:
        print(df.to_string())
        return 1, df.at[0, 'identity'], df.at[0, 'Facenet512_euclidean_l2']
```

A partir de ahora, este fichero será eliminado una vez se detecte una nueva cara y se haya añadido a la carpeta auxiliar.

```
# Different faces for embeddings directory creation
    db_images = f"{output_dir}/aux-faces"
    representations_path = f"{db_images}/representations_facenet512.pkl"
    os.mkdir(db_images)

    for image_path, image_name in images:
        # TODO
        # Face detection
        status, path, distance = face_detection(image_path, db_images,
models[2], metrics[2])
        if status == 0:
            new_path = f"{output_dir}/face_{faces_discovered_counter}"
```

```
            aux_output_path =
f"{db_images}/face_{faces_discovered_counter}{os.path.splitext(image_name)[1]}
"
            faces_discovered_counter += 1

            # Directory creation
            os.mkdir(new_path)

            # Moving the file to the new directory
            output_path = f"{new_path}/{image_name}"

            # Copy the file
            shutil.copyfile(image_path, output_path)

            # Copy the file to aux-faces
            shutil.copyfile(image_path, aux_output_path)

            if os.path.exists(representations_path):
                os.remove(representations_path)

        else:
            base=os.path.basename(path)
            output_path =
f"{output_dir}/{os.path.splitext(base)[0]}/{image_name}"
            shutil.copyfile(image_path, output_path)
```

A destacar queda la nueva estructura de directorios y rutas, pues ahora, si se encuentra una nueva cara, será añadida a la carpeta auxiliar con el nombre “face_X”, para poder saber donde guardar posteriores caras iguales a esta. Por otro lado, también esta imagen será clasificada en su nueva carpeta correspondiente, al igual que en la versión anterior.

Si no es una nueva cara, se extrae el *basename* de la imagen a la cual más se parece según DeepFace, lo cual nos dará el nombre de la carpeta donde clasificarla. Una vez obtenido, se forma la ruta y se copia allí el archivo.

Esta versión, al no comparar cada imagen con todas las imágenes ya clasificados, pierde un poco en precisión de clasificación respecto a la versión anterior, aunque gana muchísimo en velocidad, creando un equilibrio entre estos dos factores. Aun así, existe algún par modelo-métrica que arroja resultados excelentes, los cuales serán detallados más adelantes.

Análisis de resultados

Para llevar a cabo un análisis de los resultados se debe profundizar en las 3 técnicas llevadas a cabo para realizar el estudio. Estas técnicas son las siguientes:

- Empleo de embeddings
- Uso de DeepFace – Comparativa con todas las imágenes
- Uso de DeepFace – Equilibrio entre velocidad y precisión

Empleo de embeddings

Empleando la demostración BuscaParecidos vista en la práctica 6 de la asignatura como punto de partida, se intenta guardar unos embeddings de referencia por cada cara nueva y distinta detectada para comparar los embeddings obtenidos de la imagen a clasificar con los de cada cara distinta.

Este desarrollo terminó arrojando muy malos resultados, confundiendo muchas de las caras en algunos casos, o determinando que la mayoría de las caras eran distintas, por lo que, aunque como punto de partida ha permitido situarnos en el problema y explorar otras herramientas, de esta forma la aplicación, por muy veloz que sea, no cumplía con los requisitos para poder presentarla. De esta forma, no se incluye ni su código ni resultados al ser despreciada, y se pasa directamente al uso de DeepFace.

Uso de DeepFace - Comparativa con todas las imágenes

En este modelo, se intenta priorizar la precisión comparando cada imagen a clasificar con todas las imágenes ya clasificadas mediante el empleo de DeepFace. Respecto a los resultados obtenidos de las distintas pruebas realizadas, se ha escogido un total de 15 posibles técnicas de resolución, ya que se han usado varios modelos, en concreto 5, tales como Facenet512, VGG-Face, SFace, Facenet y ArcFace, y a su vez cada uno de estos modelos han sido probados con tres métricas distintas: Euclidean, Euclidean_l2 y Cosine.

El conjunto de imágenes creado para estas pruebas consta de un total de 50 fotos de personas célebres conocidas mundialmente. Esta cantidad de fotos se distribuye de manera que cada famoso tiene un total de 5 imágenes.

El resultado ideal sería que se cree un total de 10 carpetas, 1 por cada cara reconocida, y que dentro de cada carpeta tuviera un total de 5 imágenes.

Teniendo esta idea en cuenta, se procede a realizar un descarte de todos los modelos los cuales no se acerquen a este número de carpetas. A continuación, se muestran tres tablas cuya función va a ser realizar este descarte.

MÉTRICA EUCLIDEAN	
MODELO	**Nº CARAS**
facenet512	8
VGG-Face	3
SFace	12
Facenet	14
ArcFace	16

MÉTRICA EUCLIDEAN_L2	
MODELO	**Nº CARAS**
facenet512	12
VGG-Face	7
SFace	13
Facenet	21
ArcFace	11

MÉTRICA COSINE	
MODELO	**Nº CARAS**
facenet512	25
VGG-Face	4
SFace	11
Facenet	19
ArcFace	9

Al visualizar las tablas, se pueden diferenciar un total de 6 posibles conjuntos de modelo y métrica que se acercan a los valores deseados, lo cual significa que se ha descartado 9 modelos por tener una gran disconformidad con los resultados esperados.

Una vez obtenidos los resultados de estos seis posibles modelos y métricas, se procede a realizar una fase de selección de los mejores. A diferencia de la anterior fase, en esta se va a investigar más a fondo dentro de cada modelo el número de imágenes que tiene cada cara reconocida.

MÉTRICA EUCLIDEAN	
FACENET512	
CARA	**FOTOS POR CADA CARA**
1	16
2	10
3	4
4	1
5	4
6	5
7	5
8	5
SFACE	
CARA	**FOTOS POR CADA CARA**
1	20
2	2
3	2
4	2
5	5
6	3
7	4
8	1
9	4
10	2
11	1
12	4

En cuanto a los resultados obtenidos de los modelos FaceNet512 y SFace, empleando la métrica euclidean, se puede observar que, el modelo Facenet512 actúa de una manera mucho más precisa que el modelo SFace, ya que, al analizar gráficamente cada carpeta generada, se puede observar una mayor precisión a la hora de generar carpetas con las distintas caras en el primer modelo que el segundo.

Un caso a tener en cuenta han sido que ciertos famosos los agrupa en una misma carpeta y si hay alguna foto del mismo famoso, pero con algún accesorio, lo agrupa en una carpeta con una foto individual, detectando este comportamiento en ambos modelos.

MÉTRICA EUCLIDEAN_L2	
FACENET512	
CARA	**FOTOS POR CADA CARA**
1	6
2	3
3	10
4	4
5	4
6	1
7	3
8	5
9	4
10	1
11	5
12	4
ARCFACE	
CARA	**FOTOS POR CADA CARA**
1	11
2	5
3	5
4	3
5	2
6	5
7	4
8	1
9	6
10	4
11	4

Una vez realizado el análisis de los modelos FaceNet512 y ArcFace con la métrica euclidean_l2, lo que se puede comparar con los dos modelos anteriormente analizados es un comportamiento bastante similar. En este caso se repite uno de los dos modelos analizados anteriormente que es el caso de FaceNet512, ofreciendo unos resultados bastante correctos. Pero en esta ocasión, el modelo ArcFace ha llegado a dar unos resultados mucho más precisos, ha llegado a acertar un total de 3 personas famosas al 100% y el resto las confusiones que tiene son por la misma razón comentada anteriormente, que dicha persona lleve un accesorio que hace que se separe en una carpeta distinta.

MÉTRICA COSINE	
SFACE	
CARA	FOTOS POR CADA CARA
1	6
2	5
3	10
4	1
5	3
6	6
7	5
8	5
9	4
10	4
11	1
ARCFACE	
CARA	FOTOS POR CADA CARA
1	9
2	5
3	5
4	3
5	4
6	11
7	5
8	4
9	4

En este último caso lo que se puede observar al analizar los modelos SFace y ArcFace con la métrica Cosine y sus respectivas carpetas resultantes es que, ambos modelos procesan bastante bien todas las caras que se les pasa de ejemplo. En comparación con los otros conjuntos modelo-métrica y sus resultados obtenidos, se puede ver que estos son bastante precisos en comparación con los anteriores. En algunos casos se puede ver que se ha terminado de corregir el error de separación de personas según si llevan un accesorio u otro.

Un caso a destacar es el de LeBron James, jugador de Los Ángeles Lakers, que en todos los casos anteriores se le separaba en una carpeta aparte la imagen en la que lleva la equipación de color morado, y en estos dos modelos se procesa mejor este tipo de caso ya que lo agrupa todo en la misma carpeta.

Uso de DeepFace - Equilibrio entre velocidad y precisión

Para este caso lo que se va a realizar es un término intermedio entre los dos casos anteriormente explicados. Empleando DeepFace, se combina la primera idea de guardar los embeddings de cada cara nueva detectada con la segunda idea correspondiente al empleo de la herramienta Find de DeepFace. De esta forma, en vez de guardar embeddings, directamente se guarda la imagen de cada cara nueva detectada, y en vez de comparar una imagen con todas las imágenes ya clasificadas, solo se comparará con las distintas caras detectadas guardadas en una carpeta auxiliar, favoreciendo la velocidad respecto al desarrollo anterior.

MÉTRICA EUCLIDEAN	
MODELO	**Nº CARAS**
facenet512	16
VGG-Face	7
SFace	17
Facenet	20
ArcFace	20

MÉTRICA EUCLIDEAN_L2	
MODELO	**Nº CARAS**
facenet512	13
VGG-Face	12
SFace	18
Facenet	25
ArcFace	12

MÉTRICA COSINE	
MODELO	**Nº CARAS**
facenet512	24
VGG-Face	11
SFace	16
Facenet	20
ArcFace	11

En comparación con el desarrollo realizado buscando precisión, se puede observar claramente que todos los resultados en su gran mayoría han sido muy superiores a los anteriores. Esto se debe a que se ha intentado buscar velocidad también, ya que, en los casos de precisión, el tiempo promedio a realizar la separación de caras era en torno a unos 3-4 minutos, mientras que aquí se reduce a 1 minuto aproximadamente por cada caso a comprobar.

Se puede observar que para la métrica Euclidean, no hay ningún modelo que se encuentre cercano a las caras esperadas. Casi todos los resultados se encuentran por fuera

del rango que se está teniendo en cuenta que es rango de +2 o -2 caras de las esperadas que son 10. En este caso hay algunos modelos que llegan a incluso duplicar las caras esperadas llegando así la cifra a 20. Por lo cual se procede a descartar el uso de esta métrica y todos sus modelos para este tipo de análisis.

Una vez hecha la primera selección de los modelos y métricas a analizar, lo siguiente a realizar es una investigación a mayor profundidad de cada caso seleccionado.

MÉTRICA EUCLIDEAN_L2	
VGG-FACE	
CARA	**FOTOS POR CADA CARA**
1	5
2	8
3	5
4	1
5	4
6	3
7	7
8	7
9	4
10	4
11	1
12	1
ARCFACE	
CARA	**FOTOS POR CADA CARA**
1	5
2	4
3	7
4	5
5	4
6	3
7	5
8	4
9	1
10	4
11	4
12	4

Realizando la investigación de la métrica Euclidean_l2 con sus respectivos modelos, se puede observar que el modelo VGG-Face tiene un comportamiento medianamente aceptable, ya que realiza varias confusiones que no tienen mucho sentido. Algunos famosos con barba los mezcla y los agrupa en una misma carpeta como es el caso del jugador de baloncesto Stephen Curry con LeBron James.

A su vez, otro caso a destacar ha sido el hecho de mezclar jugadores como Andrés Iniesta con Russell Westbrook. Esto puede ser a causa de la forma de la cara o por el poco pelo y por ello los llega a confundir.

En cambio, el comportamiento del modelo ArcFace ha dado unos resultados muy precisos, llegando a realizar varias comprobaciones perfectas y solo llega a confundir y separar en una carpeta aparte algunos casos de fotos en donde las personas llevan algunos accesorios. Por lo cual se considera que este modelo es bastante útil para realizar búsquedas rápidas y precisas.

MÉTRICA COSINE	
VGG-FACE	
CARA	**FOTOS POR CADA CARA**
1	8
2	10
3	5
4	4
5	6
6	4
7	6
8	1
9	4
10	1
11	1
ARCFACE	
CARA	**FOTOS POR CADA CARA**
1	5
2	4
3	6
4	5
5	4
6	3
7	5
8	5
9	5
10	4
11	4

Los resultados obtenidos de la métrica Cosine han sido muy similares a los anteriores. Al tratarse de los mismos modelos, los resultados han sido bastante parejos, llegando prácticamente a las mismas conclusiones que la métrica anterior. Cabe destacar que, en algunos casos, ambos modelos confunden varios famosos y los agrupa en una misma carpeta. Este fenómeno ocurre sobre todo en el primer modelo, es decir, el modelo VGG-Face. Un ejemplo de esto último comentado ha sido la mezcla de Brad Pitt junto a Andrés Iniesta y Messi, los tres en una misma carpeta.

Sin embargo, la precisión es notablemente más alta en el modelo ArcFace, el cual parece ser el que mejor diferencia las distintas caras. Llegando a reconocer al 100% hasta a 5 famosos, y varios de ellos tienen hasta 4 imágenes en sus respectivas carpetas, lo cual es un resultado muy positivo ya que consigue esta precisión tan alta y se consigue en un periodo de tiempo relativamente corto.

Conclusiones

Una vez que se han analizado los resultados de las tres distintas variantes posibles planteadas, se puede ver que un mismo problema se puede abordar de distintas maneras, ya sea priorizando velocidad, priorizando precisión o realizando algo intermedio.

Priorizando la velocidad, mediante el uso de embeddings, no se ha conseguido obtener resultados medianamente adecuados respecto a lo esperado para este trabajo, pues no consigue clasificar bien los conjuntos de imágenes suministrados y tampoco da indicios de que pueda lograrlo.

Si se prioriza la precisión hay que tener en cuenta que el tiempo promedio a realizar cada una de las clasificaciones es de unos 3-4 minutos por cada método a utilizar. Lo bueno de este tipo de técnica es que la precisión de casi todos los modelos será decente sin dar valores excesivamente dispares a los esperados exceptuando algún que otro caso.

Las tres métricas utilizadas tienen valores bastante buenos, por lo cual se podrían considerar cualquiera de las 3 métricas para llevar a cabo dicho estudio. Los modelos que más se han repetido y mejores resultados han ofrecido para esta técnica han sido el Facenet512 ofreciendo unos resultados bastante precisos y el modelo ArcFace, cuyos resultados también han sido muy correctos incluso llegando a superar en algunos casos al Facenet512.

Si se pretende mantener un equilibrio entre velocidad y precisión, lo que se obtiene es que cada clasificación tomará un tiempo medio de 1 min aproximadamente, lo cual reduce 3 veces el tiempo anterior, también la precisión es un poco diferente para este tipo de técnica ya que, algunos modelos dan valores muy dispares a los esperados, pero otros son muy precisos lo cual es muy bueno.

A destacar de esta técnica, las métricas con mejores resultados han sido euclidean_l2 y cosine utilizando los modelos VGG-Face y ArcFace, este último es con el que mejores resultados se han obtenido.

Propuesta de ampliación

Se nos ocurren muchas propuestas de ampliación que pueden realmente dar lugar a un producto software competente. A continuación, se van incluyendo las propuestas que mejorarían progresivamente la aplicación.

En primer lugar, sería interesante que la interfaz gráfica permitiera la selección del modelo que se utilizará en una determinada ejecución.

También, como se ha explicado, hemos encontrado la necesidad de tomar decisiones que han comprometido la calidad de los resultados a favor del tiempo de ejecución a nivel algorítmico, por lo que también sería interesante disponer de una serie de algoritmos y, a través de la interfaz gráfica, seleccionar el algoritmo específico para cada ejecución. Esto se puede realizar con relativa sencillez aplicando el patrón de diseño *strategy*.

Por otra parte, se puede añadir una funcionalidad que, dada una imagen de una persona, busque, devuelva o marque todas las fotografías en las que aparece dicha persona.

Se pueden hacer cambios importantes en la interfaz gráfica, dando lugar a una aplicación mucho más completa, como un panel de navegación por los distintos directorios de un proyecto. Una vista para cada una de las funcionalidades (la implementada y la propuesta en el párrafo anterior por el momento), así como una vista que permita desarrollar algoritmos sin tener que acceder a los archivos fuente del proyecto. Dado que se propone el uso del patrón de diseño *strategy*, la agregación de nuevas estrategias es tan simple como crear un nuevo fichero con una clase que implemente la interfaz correspondiente y proporcione la funcionalidad que se quiera añadir. Para ello esta vista sería bastante sofisticada, proporcionando un editor de texto, a poder ser competente y alguna interfaz para, en caso necesario, instalar paquetes adicionales.

Indicación de herramientas/tecnologías con las que les hubiera gustado contar

De cara a la creación de la interfaz gráfica, la librería empleada, PyQt5 tiene sus limitaciones y por suerte o por desgracia estamos más acostumbrados a utilizar HTML, CSS y JavaScript por lo que habría sido interesante realizar la parte gráfica como frontend de aplicación web y la parte algorítmica como backend. Para mantener una separación entre ambos y una independencia de paquetes, lógica, repositorios y no tener que lidiar con las rutas de Windows, se pueden encapsular tanto el backend y el frontend propuestos en dos contenedores Docker que emulen sistemas Linux.

Fuentes y tecnologías utilizadas

- Python como lenguaje de programación: https://www.python.org/
- PyQt5 para la interfaz gráfica: https://pypi.org/project/PyQt5/
- DeepFace como herramienta de detección y reconocimiento: https://github.com/serengil/deepface
- https://www.geeksforgeeks.org/deep-face-recognition/
- https://www.quora.com/How-does-facial-recognition-work-in-Google-Photos-Does-it-find-similarities-with-the-same-person%E2%80%99s-childhood-photos-also
- https://viso.ai/computer-vision/deepface/

MATEUS SANCHEZ RAUL - 45369519H

Firmado digitalmente por MATEUS SANCHEZ RAUL - 45369519H
Fecha: 2023.01.08 21:25:40 Z