Proyecto 3 - Base de datos multimedia

Librería utilizada

Principales librerías utilizadas:

- RTree
- face_recognition
- pickle

Data Collection Processing

R-Tree Data Collection Processing

Se requiere de la construcción de los archivos de memoria secundaria de nuestro R-Tree con la colección de fotos prederteminada de caras conocidas, en dónde se realizaran todas las busquedas por similitud. Para ello se implemento el siguiente código:

```
def process_collection(rtree_name, n): path = base_path +
"/DataProcessing/Collection/lfw" dir_list = os.listdir(path) p = index.Property()
p.dimension = 128 p.buffering_capacity = 10 rtree_idx = index.Index(rtree_name,
properties=p) c = 0 break_fg = False images_list = [] for file_path in dir_list:
path_tmp = path + "/" + file_path img_list = os.listdir(path_tmp) for file_name in
img_list: path_aux = path_tmp + "/" + file_name img = fr.load_image_file(path_aux) # Get
face encodings for any faces in the uploaded image unknown_face_encodings =
fr.face_encodings(img) for elem in unknown_face_encodings: if c == n: break_fg = True
break coor_tmp = list(elem) for coor_i in elem: coor_tmp.append(coor_i) tmp_obj =
{"path": path_tmp, "name": file_name} rtree_idx.insert(c, coor_tmp, tmp_obj)
images_list.append((c, path_aux)) c = c + 1 if break_fg: break if break_fg: break
rtree_idx.close() print(str(c) + " images processed") return rtree_idx
```

Se hace uso de la libreria $RTree$ para el procesamiento de la colección completa de imagenes con el N de la colección como parametro de la función que a su vez será usado para la experimentación y comparación de ambos métodos de busqueda. Esta función nos retorna dos archivos con las siguientes extensiones $.dat$ e $.idx$ que corresponden a la colección de tamaño N dada.

Sequential Data Collection Processing

Por otra parte, se requiere almacenar la colección de datos pero con un pre-procesamiento que consta de la identificación de rostros. Esto se almacena en un par de vectores que contienen la dirección de la foto procesada y la cara detectada, del tamaño N de colección recibido como parametro. Como se observa en el codigo de abajo, tambien contamos con una función de serialización que obtiene el mismo N y el par de vectores de datos. Aquí se hace uso de la libreria $pickle$ para serializar ambos vectores y guardarlos en archivos binarios con su nombre respectivo al N tamaño de la colección.

```
def SimilarFaces(n,path): dir_list = os.listdir(path) faces= [] order = [] break_fg =
False it = 0 for file_path in dir_list: path_tmp = path + "/" + file_path img_list =
os.listdir(path_tmp) for file_name in img_list: path_aux = path_tmp + "/" + file_name
img = fr.load_image_file(path_aux) unknown_face_encodings = fr.face_encodings(img) for
elem in unknown_face_encodings: if it == n: break_fg = True break order.append(path_aux)
faces.append(elem) it = it + 1 if break_fg: break if break_fg: break return faces,order
def Seq_Serializer(n,a,b): path = base_path + '/DataProcessing/seq_files/' data = [a,b]
filenames = ['faces','order'] it = 0 for d in data: file = open(path+filenames[it] +
str(n) + '.dat','wb+') pickle.dump(d,file) file.close() it += 1
```

Se requiere de la construcción de los archivos de memoria secundaria de nuestro R-Tree con la colección de fotos prederteminada de caras conocidas, en dónde se realizaran todas las busquedas por similitud. Para ello se implemento el siguiente código:

```
def process_collection(rtree_name, n): path = base_path +
"/DataProcessing/Collection/lfw" dir_list = os.listdir(path) p = index.Property()
p.dimension = 128 p.buffering_capacity = 10 rtree_idx = index.Index(rtree_name,
properties=p) c = 0 break_fg = False images_list = [] for file_path in dir_list:
path_tmp = path + "/" + file_path img_list = os.listdir(path_tmp) for file_name in
img_list: path_aux = path_tmp + "/" + file_name img = fr.load_image_file(path_aux) # Get
face encodings for any faces in the uploaded image unknown_face_encodings =
fr.face_encodings(img) for elem in unknown_face_encodings: if c == n: break_fg = True
break coor_tmp = list(elem) for coor_i in elem: coor_tmp.append(coor_i) tmp_obj =
{"path": path_tmp, "name": file_name} rtree_idx.insert(c, coor_tmp, tmp_obj)
images_list.append((c, path_aux)) c = c + 1 if break_fg: break if break_fg: break
rtree_idx.close() print(str(c) + " images processed") return rtree_idx
```

Se hace uso de la libreria $RTree$ para el procesamiento de la colección completa de imagenes con el N de la colección como parametro de la función que a su vez será usado para la experimentación y comparación de ambos métodos de busqueda. Esta función nos retorna dos archivos con las siguientes extensiones $.dat$ e $.idx$ que corresponden a la colección de tamaño N dada.

Técnica de indexación

R-Tree

Esta estructura se utiliza para métodos de acceso espacial. Esta especialmente diseñada para hacer búsquedas de coordenadas en un plano. La principal desventaja de los R-Tree es su perdida de velocidad en altas dimensiones. El vector característico de una imagen es una coordenada pero en un plano de muchas dimensiones. El R-Tree es una estructura prometedora y muy rápida para trabajos en dos o tres dimensiones; por eso, la técnica a continuación es usada.

Busquedas

KNN-Sequential

La busqueda de vecinos más cercanos de manera secuencial es directa. Pasamos por todas las imagenes obteniendo su vector caracteristico calculando su distancia hacia nuestra imagen por medio de la función `face_recognition.api.face_distance`. Almacenamos las distancias en un min-heap y rescatamos al final las k primeras.

```
def KNN_Seq(k, query,n): path = base_path + '/DataProcessing/seq_files/' faces =
pickle.load(open(path+'faces'+str(n)+'.dat','rb')) order =
pickle.load(open(path+'order'+str(n)+'.dat','rb')) distances = fr.face_distance(faces,
query) res = [] for i in range(n): res.append((distances[i], order[i]))
heapq.heapify(res) result = heapq.nsmallest(k, res) return result
```

KNN-R-Tree

En el caso del R-Tree la búsqueda se puede volver más veloz pero más compleja de implementar. Bajamos desde el *root* del árbol preguntando si la mínima distancia entre la consulta y el *bounding box* es menor a la menor distancia hallada. La menor distancia hallada es la menor distancia alrededor de la consulta que por el momento se ha encontrado donde tenemos presuntamente los k vecinos más cercanos. Inicialmente este valor se encuentra en infinito. En un nodo hoja exploramos todos los elementos en donde su distancia sea menos a la menor distancia hallada. Mientras aun no hallamos los k vecinos seguiremos con la misma menor distancia hallada. Cuando ya tengamos por los k vecinos en la lista, actualizamos la menor distancia hallada a la mayor distancia entre los k puntos que hemos encontrado por el momento. Vamos visitando cada nodo hoja con la esperanza que tenga un menor valor que la menor distancia hallada, si ese fuera el caso procedemos a eliminar el punto más lejano y reemplazarlo por el que estamos visitando.

```
def NNquery(k, q, n, result): if n.isLeaf: for obj in n.points: if dist(obj, q) < result
_dist: if result.size > k: result_dist = dist(obj, q) result.remove_greatest() result.ap
pend(obj) else: for child in n.childrens: if mindist(q, child.box) < result_dist: NNquer
y(k, child)
```

Psudocodigo del algoritmo

En este no es necesario implementarlo ya que la librería *Rtree* de Python lo tiene hecho.

Por lo tanto creamos el siguiente código para la busqueda de similitud dado un k y un $query$.

```
def KNN_rtree(k, query): path = base_path + "/DataProcessing/" rtree_name = path +
'rtreeFile' p = index.Property() p.dimension = 128 # D p.buffering_capacity = 10 # M
rtreeidx = index.Rtree(rtree_name, properties=p) query_list = list(query) for query_i in
query: query_list.append(query_i) return rtreeidx.nearest(coordinates=query_list,
num_results=k, objects='raw')
```

Maldición de la dimensionalidad

Análisis

Una imagen nos produce una serie de vector característicos por cada rostro encontrado. Este vector caracteristico tiene una 128 dimensiones, por esto en el R-Tree involucra *bounding boxes* de dimensiones enormes. Esto relentiza considerable la velocidad del R-Tree por tener que aplicar alguna técnica para reducir su dimensionalidad. Existen conceptos como Space-filling curve, Principal component analysis (PCA) y estructuras de datos para altas dimensiones como X-Tree. que nos pueden ayudar a solucionar este problema. Un trade off de alguna de estas técnicas es su precisión.

Mitigación

Lamenteblemnete no hemos llegado a mitigar la maldición de dimensionalidad; sin embargo, teniamos las alternativas de PCA, Space-filling curve y X-Tree.

Experimentación

Marco experimental

Computadora usada para la experimentación

CPU: Intel i7 9700k
SSD: Samsung 960 Evo 500GB
RAM: 16GB 3200Mhz CL16

Terminos

N es el número de rostros.

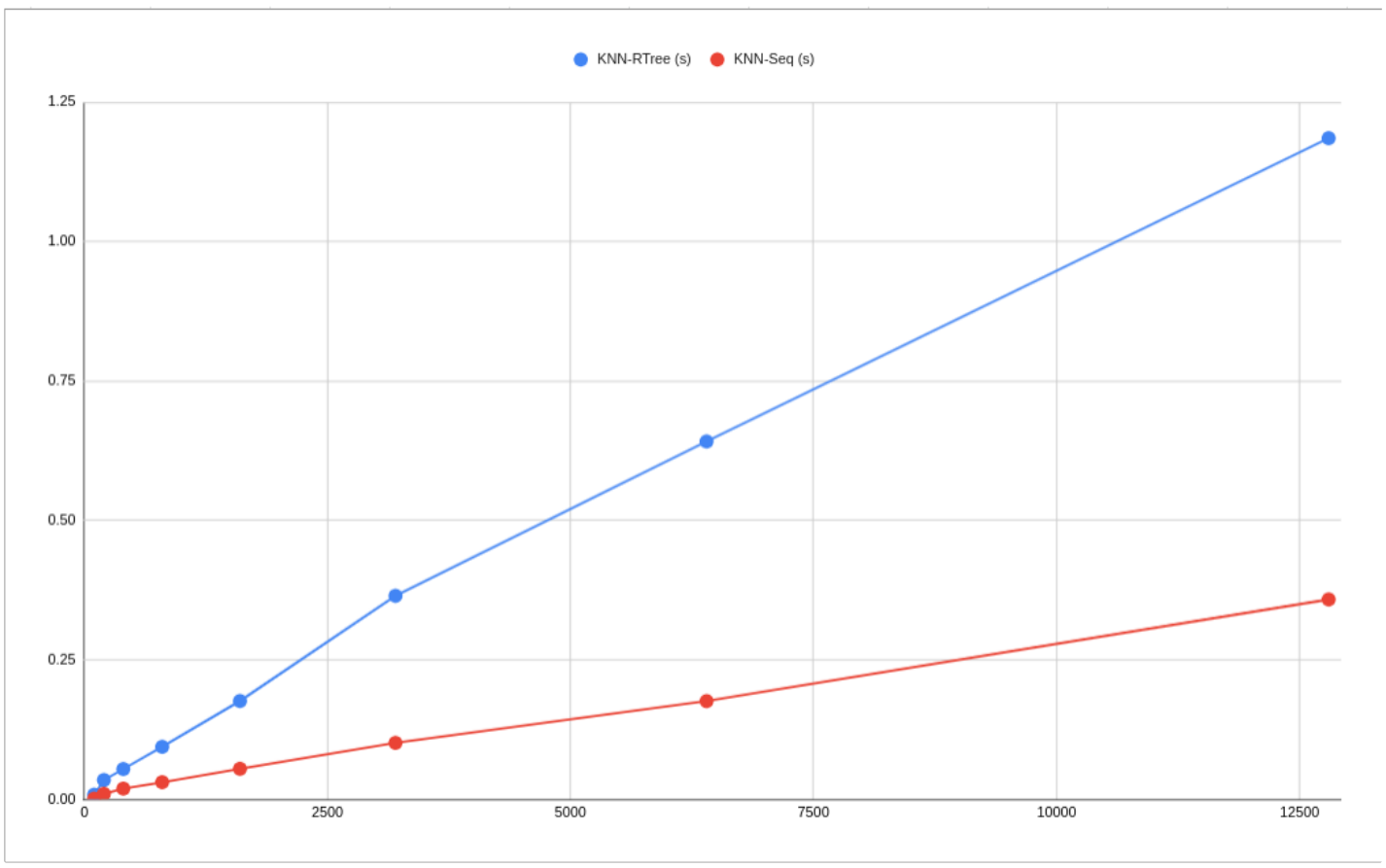
k es el número de vecinos más cercanos a buscar. En este caso buscamos los 8 vecinos más cercanos.

M es el máximo valor que puede tener un nodo en el R-Tree.

m es el mínimo valor que puede tener un hijo en el R-Tree.

Resultados

N	KNN-RTree	KNN-Seq
100	0.00801	0.00089
200	0.03430	0.00961
400	0.05401	0.01893
800	0.09388	0.03041
1600	0.17603	0.05440
3200	0.36470	0.10092
6400	0.64153	0.17582
12800	1.18549	0.35823



Análisis

Como podemos observar en el siguiente gráfico podemos ver la busqueda en KNN en el R-Tree crece muy rápidamente, esto es por la maldición de dimensionalidad. El R-Tree esta lidiando con *bounding boxes* de dimensiones muy altas. Esto deja mucho espacio y crea muchos *bounding boxes*. La busqueda secuencia es más rápida ya que se evita explorar el inmenso R-Tree.

Discusión

Es importante lidiar con la maldición de dimensionalidad para poder tener un correcto desempeño en un estructura como R-Tree. En caso que el tiempo de implementación sea limitado, hacer una busqueda secuencial puede resultar más barato que usar una estructura compleja. Se necesitara un mayor desarrollo en estrucuturas de datos y maldición de dimensionalidad para hacerle frente a los nuevos desafíos del mañana para las siguientes bases de datos multimedia.