

Visión por Computador:

Práctica 3

Indexación y Recuperación

de imágenes

Antonio Manuel Milán Jiménez



22 de diciembre de 2017

Ejercicio 1

En este primer ejercicio realizaremos un emparejamiento de los descriptores extraídos de dos imágenes, calculando las correspondencias de los puntos SIFT de una región seleccionada de una de las imágenes con todos los puntos SIFT de la segunda imagen.

Vamos a empezar incorporando la función de imprimir imágenes que hemos usado en las anteriores prácticas junto con algunas funciones proporcionadas para extraer regiones de una imagen o cargar un diccionario por ejemplo. Es importante mencionar que se ha modificado la función de “click_and_draw()” para facilitar la extracción de regiones (ahora se selecciona un punto de la región con click_derecho y se termina la selección de la región con click_izquierdo):

```
import cv2
import pickle
import numpy as np
from matplotlib import pyplot as plt
import math
from math import sqrt
from PIL import Image
from scipy import signal
import copy

#Función encargada de imprimir una lista de imágenes
def imprimir(imagenes,fila,col,titulos,lista_puntos,lista_segmentos,grises=False):

    n_img=1
    for img in imagenes:

        for puntos in lista_puntos:
            cv2.circle(img,(int(puntos[1]),int(puntos[0])),int(puntos[2]*5),(0,255,0))

        for segmento in lista_segmentos:
            cv2.line(img,segmento[0],segmento[1],(0,0,0))

    plt.subplot(fila,col,n_img)
    plt.subplots_adjust(hspace=0.8)
    if(grises==True):
        plt.imshow(img,cmap='gray')
    else:
        plt.imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB))
    plt.title(titulos[n_img-1])

    n_img=n_img+1
    plt.show()
```

```

#Función encargada de cargar un diccionario a partir de un archivo
def loadDictionary(filename):
    with open(filename,"rb") as fd:
        feat=pickle.load(fd)
    return feat["accuracy"],feat["labels"],feat["dictionary"]

#Función encargada de cargar descriptores y parches a partir de un archivo
def loadAux(filename, flagPatches):
    if flagPatches:
        with open(filename,"rb") as fd:
            feat=pickle.load(fd)
        return feat["descriptors"],feat["patches"]
    else:
        with open(filename,"rb") as fd:
            feat=pickle.load(fd)
        return feat["descriptors"]

#Función encargada de obtener los coordenadas de un recuadro de la imagen
def click_and_draw(event,x,y,flags,param):
    global refPt, imagen,FlagEND

    # if the left mouse button was clicked, record the starting
    # (x, y) coordinates and indicate that cropping is being
    # performed
    if event == cv2.EVENT_LBUTTONDOWN:
        FlagEND= False
        cv2.destroyAllWindows("image")

    elif event == cv2.EVENT_LBUTTONDOWN:
        #refPt.append((x, y))
        #cropping = True
        print("rfePt[0]",refPt[0])
        FlagEND= False
        cv2.destroyAllWindows("image")

    elif (event == cv2.EVENT_MOUSEMOVE) & (len(refPt) > 0) & FlagEND:
        # check to see if the mouse move
        clone=imagen.copy()
        nPt=(x,y)
        print("npt",nPt)
        sz=len(refPt)
        cv2.line(clone,refPt[sz-1],nPt,(0, 255, 0), 2)
        cv2.imshow("image", clone)

```

```

cv2.waitKey(0)

elif event == cv2.EVENT_RBUTTONDOWN:
    # record the ending (x, y) coordinates and indicate that
    # the cropping operation is finished
    refPt.append((x, y))
    #cropping = False
    sz=len(refPt)
    print("refPt[sz]",sz,refPt[sz-1])
    cv2.line(imagen,refPt[sz-2],refPt[sz-1],(0, 255, 0), 2)
    cv2.imshow("image", imagen)
    cv2.waitKey(0)

```

#Función encargada de extraer una región de una imagen

```

def extractRegion(image):
    global refPt, imagen,FlagEND
    imagen=image.copy()
    # load the image and setup the mouse callback function
    refPt=[]
    FlagEND=True
    #image = cv2.imread(filename)
    cv2.namedWindow("image")
    # keep looping until the 'q' key is pressed
    cv2.setMouseCallback("image", click_and_draw)
    #
    while FlagEND:
        # display the image and wait for a keypress
        cv2.imshow("image", image)
        cv2.waitKey(0)
    #
    print('FlagEND', FlagEND)
    refPt.pop()
    refPt.append(refPt[0])
    cv2.destroyWindow("image")
    return refPt

```

También vamos a incorporar una función para obtener los matches encontrados entre dos imágenes usando “**Knn**” que ya programamos para la anterior práctica:

#Función que calcula los matches entre dos imágenes

```

def CalculaMatches(mi_imagen1,mi_imagen2, mascara, keypoints1, keypoints2):

    sift=cv2.xfeatures2d.SIFT_create()

    keypoints1, descriptores1=sift.detectAndCompute(mi_imagen1,mascara)
    keypoints2, descriptores2=sift.detectAndCompute(mi_imagen2,None)

```

```

matcherKnn=cv2.BFMatcher()
matchesKnn=matcherKnn.knnMatch(descriptores1,descriptores2,k=2)

mejores=[]
for a,b in matchesKnn:
    if (a.distance < 0.7*b.distance): #Lo escogemos sólo si es un 30% que el segundo mejor match.
        mejores.append(a)

mejores = sorted(mejores, key=lambda x:x.distance)
img_knn = cv2.drawMatches(mi_imagen1,keypoints1,mi_imagen2,keypoints2,mejores[0:100],N
one,flags=2)

return img_knn

```

Vamos a empezar leyendo dos imágenes de la misma escena para que tengan elementos en común. Las imágenes son 128 y 130:

```

mi_img1 = cv2.imread("imagenes/128.png")
mi_img2 = cv2.imread("imagenes/130.png")

```



Ilustración 1: Imagen 128



Ilustración 2: Imagen 130

A continuación, extraemos de la primera imagen la región relativa al futbolín pues es lo suficientemente característica para que encuentre suficientes descriptores. Para ello utilizamos “extractRegion()”:

```
points=extractRegion(mi_img1)
```



Ilustración 3: Región del futbolín en la imagen 128

Una vez delimitada la región, construimos una máscara para esa región. Para ello, construimos una matriz de ceros del mismo tamaño que la imagen y, a continuación, mediante la función de OpenCV “fillConvexPoly()” le indicamos que ponga a 1 aquellos puntos de nuestra nueva máscara que estén dentro de la región que habíamos delimitado:

```
mask=np.zeros((mi_img1.shape[0],mi_img1.shape[1],3))
mask=cv2.fillConvexPoly(mask,np.array(points,dtype=np.int32),(1.0,1.0,1.0))
```

Ya entonces, sólo queda llamar a la función “CalculaMatches()” para que calcule los matches entre las dos imágenes, proporcionándole la máscara construida para que se centre sólo en la región delimitada para la primera imagen. Este es el resultado:

```
img_knn=CalculaMatches(mi_img1,mi_img2,np.array(mask[:, :, 0],dtype=np.uint8),[],[])
imprimir([[img_knn],1,1,['Futbolin 128-130'],[],[]])
```

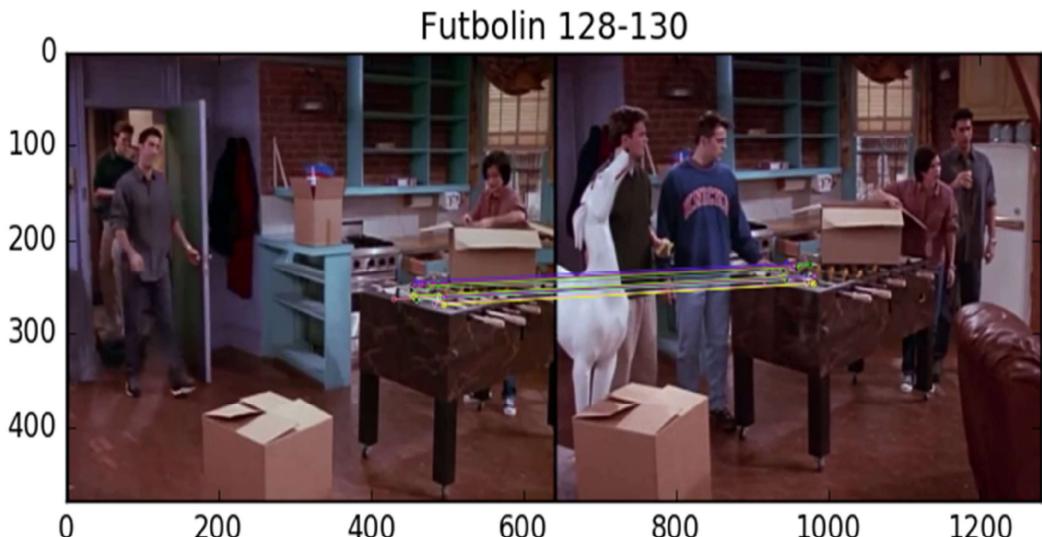


Ilustración 4: Match futbolín en imágenes 128 y 130

Vemos que ha conseguido encontrar los matches correctos para los descriptores de dentro de la región del futbolín. No son todos los matches posibles pues utilizando “Knn” estamos pidiendo que un match sea al menos un 30% mejor que el resto para incluirlo. Por lo tanto, no estamos incluyendo todos los matches, pero los que incluimos es bastante probable que sean los correctos.

Hemos realizado el mismo proceso para las imágenes 156 y 157:



Ilustración 5: Imagen 156



Ilustración 6: Imagen 157

Para la región de la cafetera y el reloj:



Ilustración 7: Región reloj imagen 156

Y éste es el resultado, donde vemos que nuevamente se han encontrado los matches correctos entre las dos imágenes para la región que habíamos delimitado:

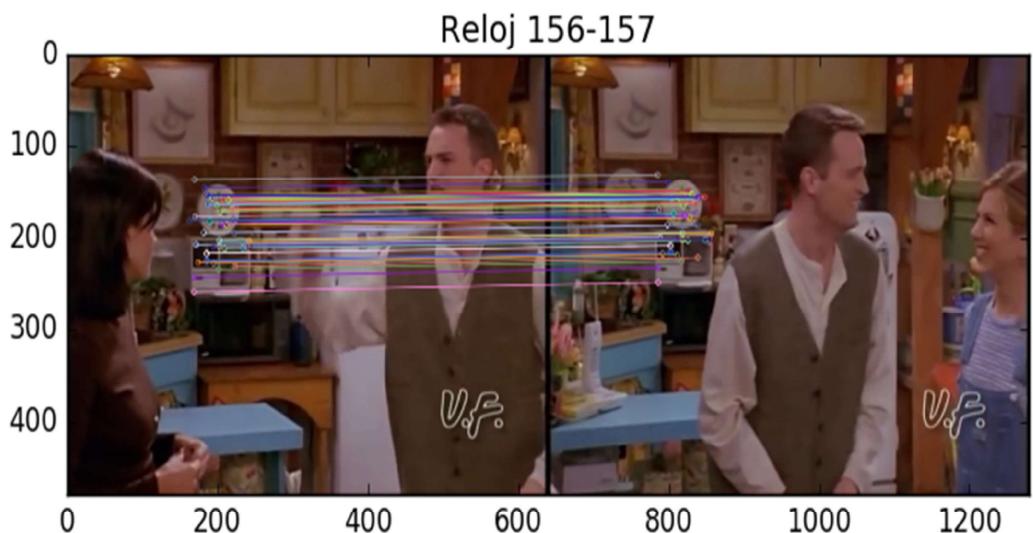


Ilustración 8: Match reloj entre imagen 156 y 157

Por último, vamos a probar con una pareja de imágenes (56 y 60) que son de la misma escena, pero la cámara está en una posición bastante diferente:



Ilustración 9: Imagen 56



Ilustración 10: Imagen 60

Para la región relativa al florero:



Ilustración 11: Región florero en la imagen 56

Y estos son los matches encontrados:



Ilustración 12: Match florero en imágenes 56 y 60

Vemos que se han encontrado muy pocos matches ya que estamos ajustando mucho el Knn para que estemos casi seguros de que los matches que encuentre sean los correctos. No obstante, estamos consiguiendo buenos matches para dos imágenes entre las que ha habido una modificación importante en la posición y ángulo de la cámara; además de un cambio importante en la intensidad lumínica que hay en la región del florero.

Si hacemos que no ajuste tanto el Knn y que incluya el match con que sea un 10% mejor que el resto obtenemos:

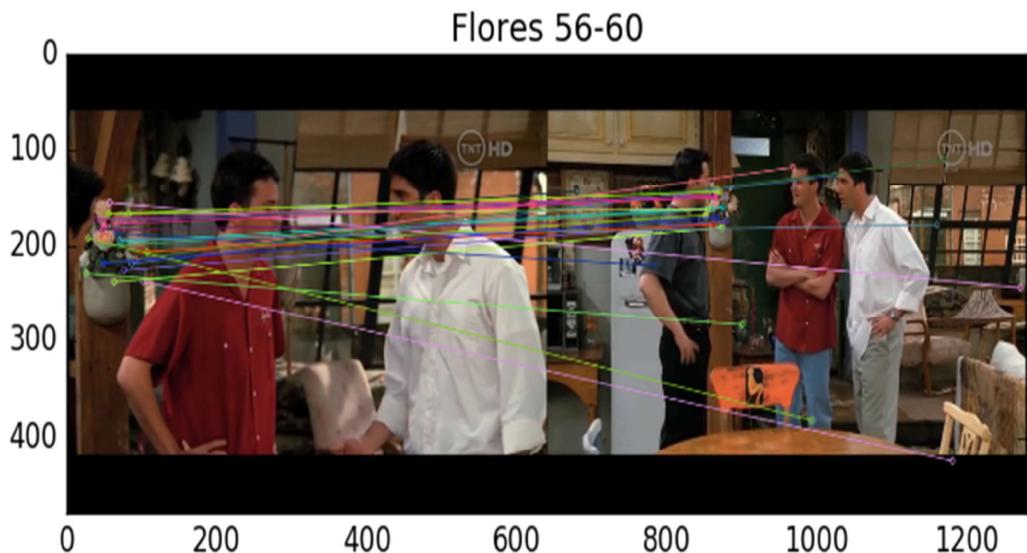


Ilustración 13: Match florero entre imágenes 56 y 60 para un filtro de 0.9

Como conclusión podemos decir que si queremos encontrar muchos y buenos matches entre dos imágenes para una región, la región escogida tiene que ser lo suficientemente característica (esto es que se detecten en ella suficientes descriptores) y que no haya un fuerte cambio en la cámara. Esto no quiere decir que no podamos hacer buenos matches entre dos imágenes que hayan sufrido deformaciones, pero sí que quizás tengamos que sacrificar la cantidad de matches para poder mantener la calidad de ellos.

Respecto a la región con suficientes descriptores, de hecho, si escogemos una región relativa al suelo o la pared, donde al ser regiones planas no vamos a encontrar suficientes descriptores (una pobre detección de gradientes), no obtendremos ningún match pues habrá demasiada ambigüedad en si los matches que encuentra son correctos o nos estamos confundiendo con otras regiones también planas, por lo que Knn los desechará.

Ejercicio 2

En este ejercicio tenemos que, a partir de los descriptores, parches y centroides proporcionados; obtener aquellos parches para un cierto centroide que sean lo suficientemente semejantes entre ellos.

Para obtener los descriptores, parches, labels y centroides simplemente cargamos el archivo “descriptorsAndpatches.pkl” y el archivo “kmeanscenters5000.pkl” proporcionados:

```
descriptors,patches=loadAux("descriptorsAndpatches.pkl",1)
compactness,labels,centers=loadDictionary("kmeanscenters5000.pkl")
```

A continuación, agruparemos los descriptores en función de los centroides que tengan asignados gracias a los datos “labels”. Estas etiquetas, que se corresponden 1-1 con los descriptores, indican el número del centroide que tienen asignado. Esta es la función que se encarga de ello:

```
#Función que para cada clase tendremos sus descriptores correspondientes
def clasificarDescriptores(labels,descriptors):
```

```
    clasificados=[]
    for i in range(5000):
        clasificados.append([])

    for i in range(len(labels)):
        tupla=(descriptors[labels[i,0]],i)
        clasificados[labels[i,0]].append(tupla)

    return clasificados
```

Recorremos las etiquetas; para cada una de ellas creamos una tupla con el descriptor correspondiente y la posición de éste, para finalmente añadirla a una lista que lleva todos los descriptores correspondientes a un centroide. Tendremos al final una lista de 5000 listas; en cada una de ellas los descriptores asociados al centroide de la posición de dicha lista.

Esta es la llamada que realizamos de la función:

```
descriptorsClasificados=clasificarDescriptores(labels,descriptors)
```

El siguiente paso es ordenar los descriptores, pues queremos quedarnos sólo con los 20 más cercanos a sus respectivos centroides. Esta es la función encargada de ordenarlos y quedarse con los 20 más cercanos, junto con su llamada correspondiente:

```
#Función que devuelve los patches más cercanos a cada uno de los centros
def obtenerCercanos(descriptors,centers,patches,tamCenters):
```

```
for i in range(len(descriptors)):
    for j in range(len(descriptors[i])):
        tupla=descriptors[i][j]
        dist=np.linalg.norm(tupla[0] - centers[i])
        descriptors[i][j]=descriptors[i][j]+(dist,)
    descriptors[i]=sorted(descriptors[i],key=lambda tup: tup[2])
    descriptors[i]=descriptors[i][0:20]

clasicadosPatches=[]
for i in range(tamCenters):
    clasicadosPatches.append([])

for i in range(len(clasicadosPatches)):
    for j in range(len(descriptors[i])):
        clasicadosPatches[i].append(patches[descriptors[i][j][1]])

return descriptors,clasicadosPatches
```

```
descriptorsCercanos,patchesCercanos=obtenerCercanos(descriptorsClasificados,centers,patches,
len(centers))
```

Para cada una de las 5000 listas, calculamos las distancias (mediante `np.linalg.norm`) entre cada uno de sus descriptores y el centroide al que están asociados. Gracias a estas distancias calculadas, mediante la función “sorted” ordenamos los descriptores de cada una de estas listas en función de las distancias al centroide correspondiente, siendo los primeros los de menor distancia. Entonces, nos quedamos con los 20 primeros.

Una vez hecho esto, también queremos tener los parches ya clasificados para poder imprimirlas fácilmente cuando sepamos aquellos que nos interesan. Vemos entonces que creamos una estructura similar a la de las listas de descriptores, donde por cada descriptor, ya ordenado, añadimos el parche correspondiente a esta nueva estructura.

A continuación, tenemos que seleccionar los descriptores asociados a un centroide que más se parezcan entre ellos, para luego imprimir los parches más similares entre ellos. Para ello, calculamos y almacenamos la varianza entre los descriptores de dentro de un centroide. Así, el centroide para el que tengamos una menor varianza, significará que sus descriptores asociados serán los más similares entre ellos. Esta es la función encargada junto con su llamada en el “main”:

```

#Función encargada de calcular las varianzas entre los descriptores
def obtenerVarianzas(descriptorsCercanos):

    for i in range(len(descriptorsCercanos)):
        for j in range(len(descriptorsCercanos[i])):
            descriptorsCercanos[i][j]=descriptorsCercanos[i][j][0]

    varianzas=np.zeros(len(descriptorsCercanos))
    for i in range(varianzas.shape[0]):
        varianzas[i]=np.mean(np.var(descriptorsCercanos[i],axis=0))

    return varianzas

varianzas=obtenerVarianzas(descriptorsCercanos)

```

Una vez que tenemos calculada la varianza entre los descriptores para cada uno de los centroides, nos quedamos con los 2 centroides con una menor varianza en sus descriptores, pues de esos serán los parches que tengamos que mostrar. Puntualizar que, dado que queremos llegar a imprimir 20 parches por cada centroide, cogeremos las 2 menores varianzas que por lo menos se hayan calculado con 5 descriptores, para que por lo menos podamos imprimir 5 parches para un centroide, a la par que no hagamos un sesgo muy fuerte si necesitásemos 20 descriptores. Así es como hemos construido la función y la llamamos en el “main”:

```

#Función encargada de obtener los centroides que hayan tenido una mejor varianza
def obtenerTop(varianzas,descriptorsCercanos,n):

    top=[]
    while len(top)<n:
        ind=np.argmin(varianzas)
        varianzas[ind]=1000000000000000
        if(len(descriptorsCercanos[ind]) > 4):
            top.append(ind)

    return top

```

```
top=obtenerTop(varianzas,descriptorsCercanos,2)
```

Gracias a que anteriormente habíamos guardado una estructura con los parches más cercanos para cada centroide, ya sólo tenemos que imprimir los 20 parches más cercanos a los 2 centroides que han tenido una menor varianza en sus descriptores:

```

lista_imgs1=[]
lista_imgs2=[]
titulos1=[]
titulos2=[]
for i in range(2):
    for j in range(20):
        if(i==0):
            lista_imgs1.append(patchesCercanos[top[i]][j])
            titulos1.append("Centroide " + str(top[i]) + " Parche " + str(j))
        else:
            lista_imgs2.append(patchesCercanos[top[i]][j])
            titulos2.append("Centroide " + str(top[i]) + " Parche " + str(j))

```

Y aquí tenemos el resultado para el mejor centroide en función de la varianza:

```
imprimir(lista_imgs1,4,titulos1,[],[])
```

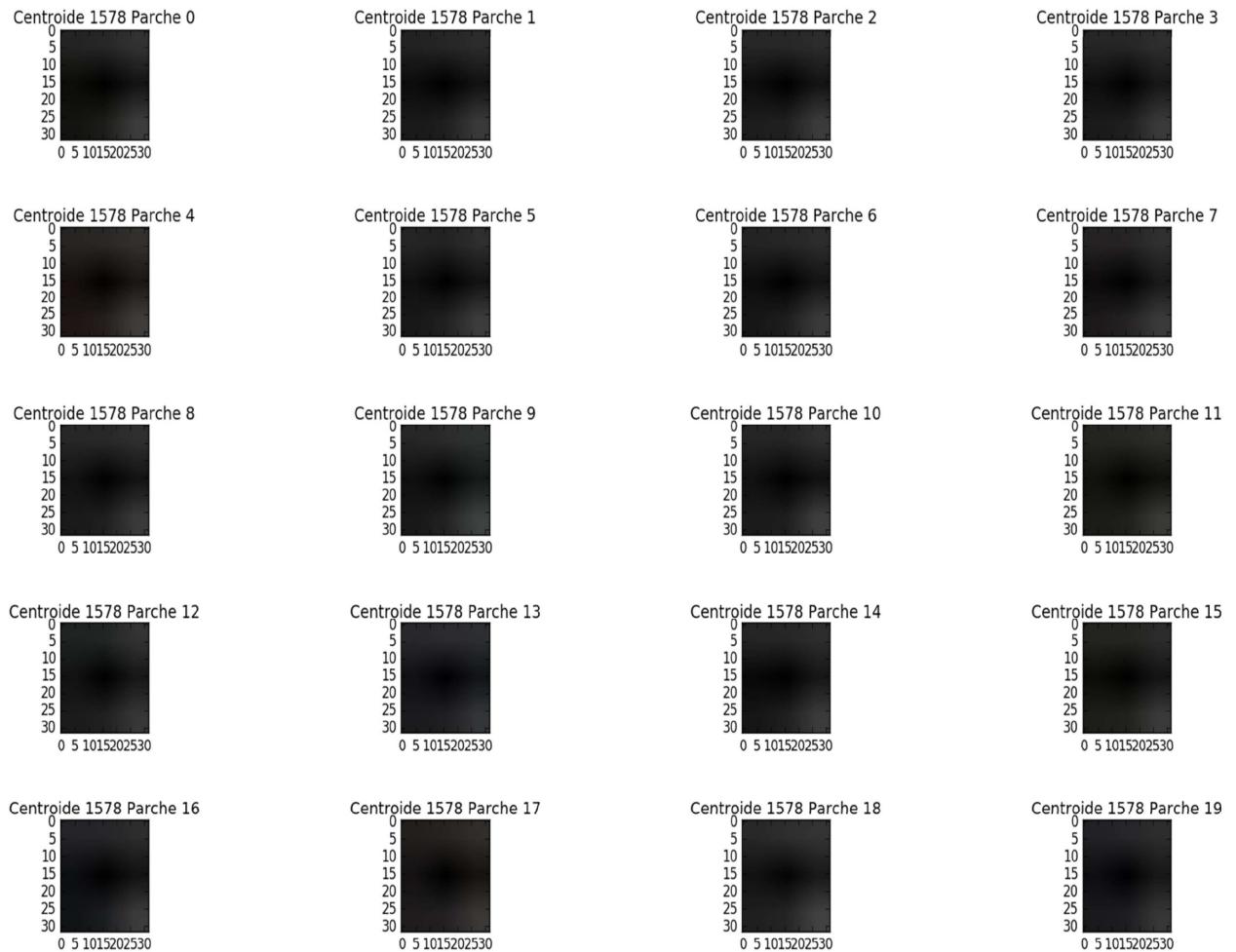


Ilustración 14: 20 parches más cercanos al centroide 1578

Y aquí para el segundo:

```
imprimir(lista_imgs2,5,4,titulos2,[],[])
```

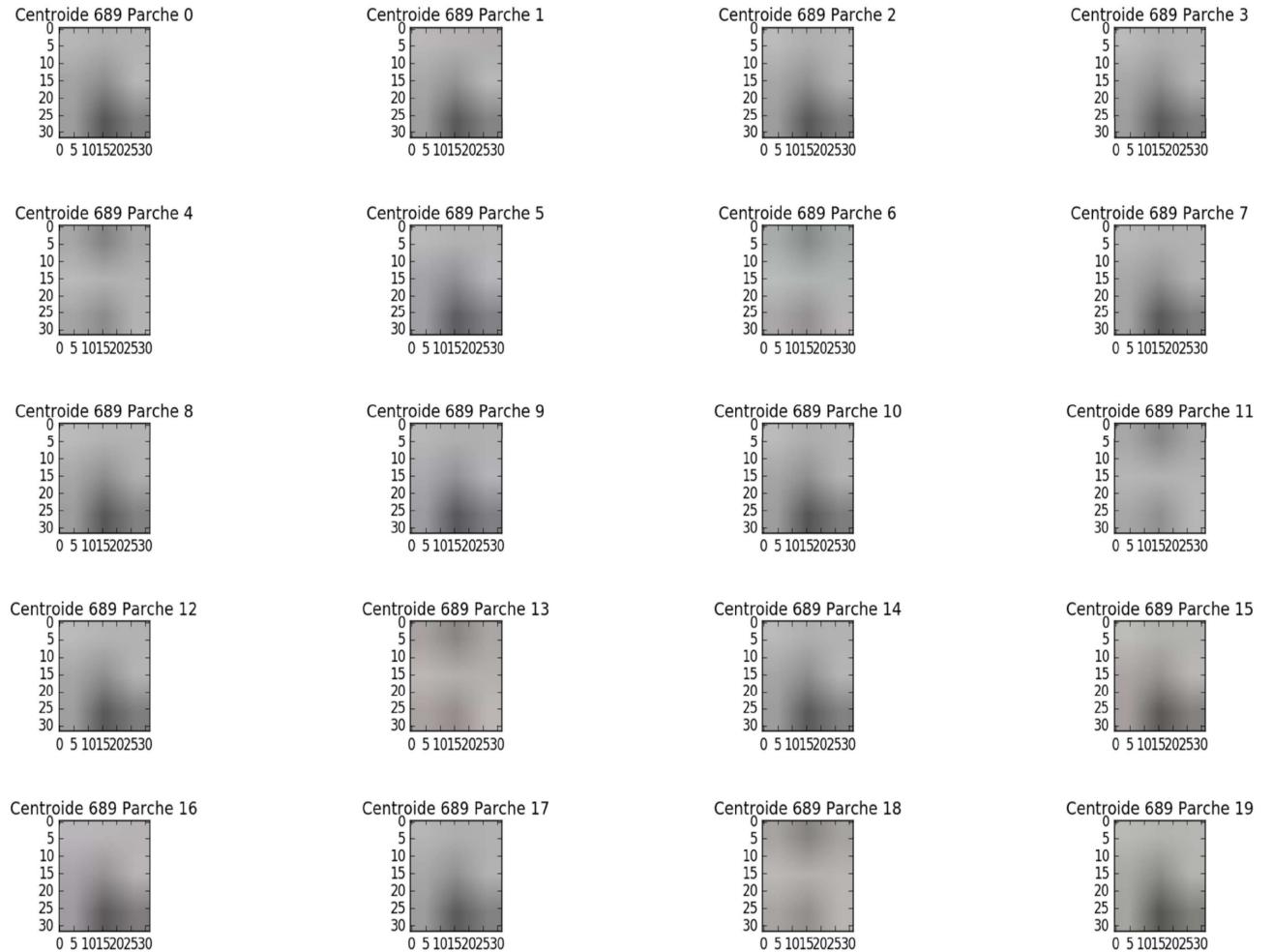


Ilustración 15: 20 parches más cercanos al centroide 689

Para los dos mejores centroides vemos que se han conseguido parches muy similares entre ellos. La diferencia a lo largo de los 20 parches es mínima. Así, vemos que hemos conseguido agrupar los descriptores por centroide y que la diferencia entre ellos muy pequeña, incluso en otros centroides.

Esto no quiere decir que los descriptores estén representando bien al centroide asignado, pues aquí solo estamos midiendo la similitud entre parches, aunque sí es verdad que son los parches más cercanos a su centroide.

Dado que estamos trabajando con un vocabulario de 5000 palabras, es posible que algunos centroides con sus respectivos descriptores se hayan dividido para poder alcanzar el número de centroides. Esto explicaría que tuviésemos dos centroides cuyos parches sean muy similares los unos a los otros.

También si disminuimos el vocabulario, el número de centroides, encontraremos ya parches que no sean muy similares los unos a los otros aunque estén para el mismo centroide. Esto es porque al haber reducido el número de centroides, aunque haya parches que no sean similares, al final se quedan asignados al mismo centroide pues no hay muchos más centroides a los que puedan asignarse. Aquí vemos un ejemplo al utilizar un vocabulario de tan sólo 500 palabras:

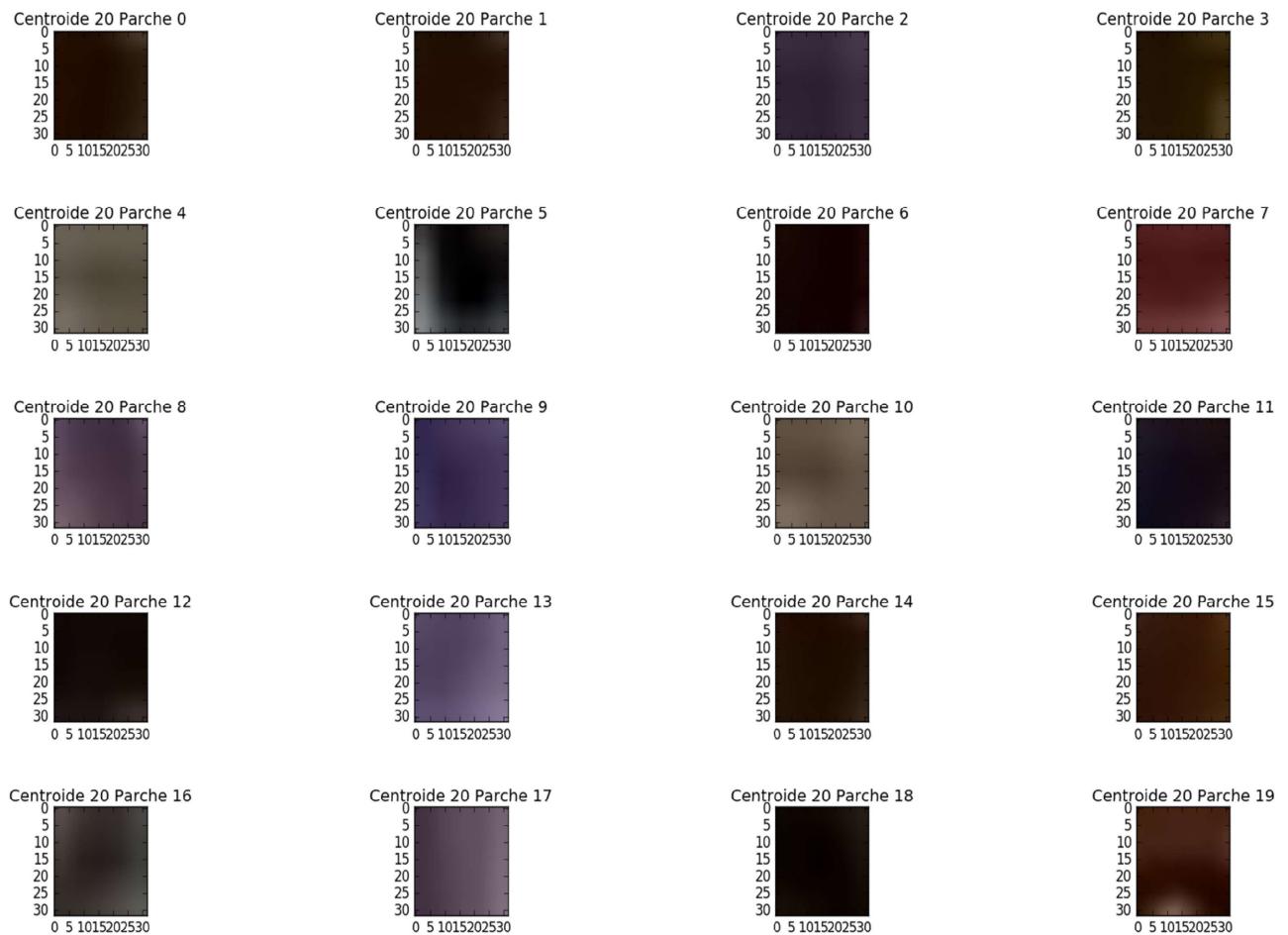


Ilustración 16: 20 parches más cercanos al centroide 20

Ejercicio 3

En este ejercicio tenemos que conseguir, mediante un índice invertido y una bolsa de palabras, que dada una imagen, seamos capaces de recuperar imágenes que pertenezcan a la misma escena.

Para empezar, cargaremos nuestro diccionario con “loadDictionary()” y añadiremos todas las imágenes del directorio a una lista de imágenes mediante la función “cargarImagenes()” que hemos construido:

#Función encargada de cargar en una lista las imágenes de un directorio
def cargarImagenes(tam):

```
    imagenes=[]

    for i in range(tam):
        s="imagenes/" + str(i) + ".png"
        img=cv2.imread(s)
        imagenes.append(img)

    return imagenes
```

```
diccionario=loadDictionary("kmeanscenters5000.pkl")
diccionario=diccionario[2]
```

```
imagenes=cargarImagenes(227)
```

El siguiente paso será construir un histograma para cada imagen en el que contaremos el número de apariciones de una palabra en la imagen. Esta es la función encargada de ello a la que le proporcionamos la lista de imágenes y el diccionario:

#Función encargada de obtener los histogramas de un conjunto de imágenes
def obtenerHistogramas(imagenes,diccionario):

```
sift=cv2.xfeatures2d.SIFT_create(nfeatures=0,nOctaveLayers=4,
contrastThreshold=0.0000000000000001)
lista_histogramas=[]

for mi_img in imagenes:
    keypoints, descriptores=sift.detectAndCompute(mi_img,None)
    nuevaM=np.dot(descriptores,np.transpose(diccionario))

    denominadores=np.apply_along_axis(np.linalg.norm,1,diccionario)
```

```

for i in range(nuevaM.shape[1]):
    nuevaM[:,i]=nuevaM[:,i]/denominadores[i]

histograma=np.zeros((5000))

for descriptor in nuevaM:
    mejor_palabra=np.argmax(descriptor)
    histograma[mejor_palabra]=histograma[mejor_palabra]+1

lista_histogramas.append(histograma)

return lista_histogramas

```

Empezamos creando un objeto “SIFT” mediante la función de OpenCV “xfeatures2d.SIFT_create()”. Dado que tenemos 5000 palabras en el vocabulario y queremos tener suficientes descriptores para poder construir buenos histogramas que no sean sólo ceros y unos, le indicamos que construya 4 capas junto con un umbral muy bajo para que consiga suficientes características.

A continuación, para cada una de las imágenes realizamos lo siguiente:

Con el objeto “SIFT” creado, obtenemos los keypoints y descriptores de la imagen. Mediante una multiplicación de matrices, “np.dot()”, multiplicamos los descriptores obtenidos por la traspuesta del diccionario. Con esta matriz resultante, tendremos la simetría de todos los descriptores encontrados en la imagen con todas las palabras del vocabulario.

En las siguientes sentencias que realizamos:

```

denominadores=np.apply_along_axis(np.linalg.norm,1,diccionario)
for i in range(nuevaM.shape[1]):
    nuevaM[:,i]=nuevaM[:,i]/denominadores[i]

```

Estamos realizando una normalización de los valores obtenidos en la matriz tal y como se indica en las transparencias de clase, mediante la fórmula:

$$sim(d_j, q) = \frac{(d_j, q)}{\|d_j\| * \|q\|}$$

Es necesario realizar una normalización ya que para comparar descriptores y palabras, necesitamos que todos estén en una única escala. Sin embargo, dado que posteriormente vamos a obtener el valor de simetría más alto por cada fila, no nos es necesario dividir por $\|d_j\|$ pues estaremos dividiendo la fila por el mismo valor, no teniendo relevancia entonces al obtener el máximo de la fila.

Ya el último paso es construir el histograma de la imagen. Para ello, recorremos por filas la matriz que hemos obtenido y normalizado. Para cada fila, descriptor, obtenemos el índice del máximo valor de simetría. Este índice representa la palabra a la que más se asemeja el descriptor en cuestión. Entonces, en nuestro histograma de palabras de la imagen, sumamos un 1 en la palabra correspondiente al índice máximo, significando que hemos encontrado una vez más esa palabra en la imagen. Así es como vamos construyendo el histograma. Una vez que tenemos el histograma construido, lo añadimos a la lista de histogramas de todas las imágenes que será la que finalmente devolvamos.

El siguiente paso es construir el índice invertido a partir de la lista de histogramas que acabamos de construir. Este índice invertido nos permitirá rápidamente saber para una palabra, todas las imágenes que contienen al menos una vez esa palabra. La función que lo construye es la siguiente:

```
#Función encargada de construir el indice invertido a partir de una lista de histogramas
def construirIndiceInvertido(lista_histogramas):
```

```
indice_invertido=[]
for i in range(5000):
    indice_invertido.append([])

indice_img=0
for histograma in lista_histogramas:
    for i in range(histograma.shape[0]):
        if(histograma[i]>0):
            indice_invertido[i].append(indice_img)
    indice_img=indice_img+1

return indice_invertido
```

En la función recorremos la lista de histogramas que habíamos calculado, uno por imagen. En cada histograma, vamos viendo aquellos índices (palabras) cuyo valor es mayor que 0 (aparece al menos una vez). Entonces, en estos índices (palabras en el índice invertido) añadimos el índice del histograma que estamos tratando en ese momento, o lo que es lo mismo, la imagen en la que hemos visto que aparecen al menos una vez estas palabras.

Al final del todo, tendremos nuestro índice invertido: para cada palabra, todas las imágenes en la que ésta haya aparecido.

A partir de este punto, ya podemos ver para una imagen cualquiera, imágenes de la misma escena. Todo el proceso necesario lo recogemos en la siguiente función:

```
#Función que a partir de una imagen, encuentra en un diccionario imágenes similares
```

```
def buscar_similares(img,diccionario, indice_invertido, lista_histogramas):
```

```
#Construimos el histograma de nuestra imagen
lista_histogramas_unico=obtenerHistogramas([img],diccionario)
histograma=list(lista_histogramas_unico[0])
```

```

#Obtenemos aquellas listas de imágenes del índice invertido para las palabras que hemos encontrado en nuestra imagen
imagenes_similares=[]
for i in range(histograma.shape[0]):
    if(histograma[i]>0):
        imagenes_similares.append(indice_invertido[i])

#Construimos una especie de histograma contando el número de apariciones de cada imagen
imagenes_finales=np.zeros(227)
for img_sim in imagenes_similares:
    for i in range(len(img_sim)):
        img=img_sim[i]
        imagenes_finales[img]=imagenes_finales[img]+1

#Construimos un "top100" con aquellas imágenes que más hayan aparecido, es decir, aquellas que más palabras tienen en común con nuestra imagen
top100=[]
imagenes_finales_aux=copy.deepcopy(imagenes_finales)
for i in range(100):
    ind=np.argmax(imagenes_finales_aux)
    top100.append(ind)
    imagenes_finales_aux[ind]=-1

#Obtenemos las simetrías entre cada uno de los histogramas del "top100" y el histograma de nuestra imagen
simetrias=np.zeros(100)
for i in range(100):
    simetrias[i]=np.dot(histograma,lista_histogramas[top100[i]])
    denom=np.linalg.norm(histograma)*np.linalg.norm(lista_histogramas[top100[i]])
    simetrias[i]=simetrias[i]/denom

#Obtenemos las 6 imágenes con mejores simetrías
top6=[]
for i in range(6):
    ind=np.argmax(simetrias)
    top6.append(top100[ind])
    simetrias[ind]=-1

return top6

```

Analizándola paso a paso, primero obtenemos el histograma de la imagen-pregunta mediante la función “obtenerHistogramas()” que describimos anteriormente.

```
lista_histogramas_unico=obtenerHistogramas([img],diccionario)
histograma=lista_histogramas_unico[0]
```

A continuación, a partir del histograma y del índice invertido que construimos anteriormente, extraemos todas las imágenes que aparecen en el índice invertido para las palabras que aparecen en el histograma de la imagen-pregunta que hayan aparecido alguna vez en la imagen.

```
imagenes_similares=[]
for i in range(histograma.shape[0]):
    if(histograma[i]>0):
        imagenes_similares.append(indice_invertido[i])
```

Con todas estas imágenes extraídas, construimos un histograma donde contamos el número de apariciones de las imágenes en esta lista:

```
imagenes_finales=np.zeros(227)
for img_sim in imagenes_similares:
    for i in range(len(img_sim)):
        img=img_sim[i]
        imagenes_finales[img]=imagenes_finales[img]+1
```

A partir de este histograma que acabamos de construir, sacamos las 100 imágenes que más hayan aparecido. Así, sólo tendremos que limitarnos a comparar los histogramas de las 100 imágenes que más palabras tengan en común con nuestra imagen-pregunta:

```
top100=[]
imagenes_finales_aux=copy.deepcopy(imagenes_finales)
for i in range(100):
    ind=np.argmax(imagenes_finales_aux)
    top100.append(ind)
    imagenes_finales_aux[ind]=-1
```

Ahora calculamos las simetrías de los histogramas de estas 100 imágenes con el histograma de nuestra imagen-pregunta mediante el producto escalar normalizado:

```
simetrias=np.zeros(100)
for i in range(100):
    simetrias[i]=np.dot(histograma,lista_histogramas[top100[i]])
    denom=np.linalg.norm(histograma)*np.linalg.norm(lista_histogramas[top100[i]])
    simetrias[i]=simetrias[i]/denom
```

Ya sólo tenemos que extraer las 5 imágenes cuyos histogramas tengan la mayor simetría respecto al histograma de la imagen-pregunta. Estas serán las 5 imágenes que devolveremos cómo las 5 imágenes que más tienen en común con la imagen-pregunta y que deberían pertenecer a la misma escena. Vemos, sin embargo, que estamos aquí obteniendo 6 imágenes, esto es porque la primera imagen que nos devolverá será la propia imagen-pregunta pues la simetría entre los histogramas será 1. Por lo tanto, estamos devolviendo la propia imagen-pregunta y otras 5 imágenes:

```
top6=[]
for i in range(6):
    ind=np.argmax(simetrias)
    top6.append(top100[ind])
    simetrias[ind]=-1

return top6
```

Con esta función ya tendremos para cualquier imagen, las 5 imágenes con las que más tiene en común y que deberían pertenecer a la misma escena. Estas son las llamadas que hacemos en el “main” de las diferentes funciones que hemos comentado en este ejercicio:

```
diccionario=loadDictionary("kmeanscenters5000.pkl")
diccionario=diccionario[2]

imagenes=cargarImagenes(227)

lista_histogramas=obtenerHistogramas(imagenes,diccionario)

indice_invertido=construirIndiceInvertido(lista_histogramas)

#Probamos con 3 imágenes-pregunta
img1=cv2.imread("imagenes/95.png")
img2=cv2.imread("imagenes/15.png")
img3=cv2.imread("imagenes/200.png")
top6_1=buscar_similares(img1,diccionario,indice_invertido, lista_histogramas)
top6_2=buscar_similares(img2,diccionario,indice_invertido, lista_histogramas)
top6_3=buscar_similares(img3,diccionario,indice_invertido, lista_histogramas)

lista_imgs1=[]
lista_imgs1.append(img1)
titulos1=[]
titulos1.append("Imagen Query " + str(top6_1[0]))

lista_imgs2=[]
lista_imgs2.append(img2)
titulos2=[]
titulos2.append("Imagen Query " + str(top6_2[0]))
```

```

lista_imgs3=[]
lista_imgs3.append(img3)
titulos3=[]
titulos3.append("Imagen Query " + str(top6_3[0]))

for i in range(3):
    for j in range(6):
        if(i==0):
            lista_imgs1.append(imagenes[top6_1[j]])
            titulos1.append("Imagen " + str(top6_1[j]))
        elif(i==1):
            lista_imgs2.append(imagenes[top6_2[j]])
            titulos2.append("Imagen " + str(top6_2[j]))
        else:
            lista_imgs3.append(imagenes[top6_3[j]])
            titulos3.append("Imagen " + str(top6_3[j]))

imprimir(lista_imgs1,3,3,titulos1,[],[])
imprimir(lista_imgs2,3,3,titulos2,[],[])
imprimir(lista_imgs3,3,3,titulos3,[],[])

```

Y aquí tenemos el resultado para las imágenes pregunta 95, 15 y 200:



Ilustración 17: Imagen query 95 y 5 imágenes encontradas para la misma escena



Ilustración 18: Imagen query 15 y 5 imágenes encontradas para la misma escena



Ilustración 19: Imagen query 200 y 5 imágenes encontradas para la misma escena

Vemos que para las dos primeras imágenes (95 y 15) obtenemos unos resultados realmente buenos pues todas o casi todas las imágenes que encontramos pertenecen a la misma escena.

Sin embargo, para la tercera imagen (200), no obtenemos un buen resultado, ninguna de las imágenes encontradas pertenece a la misma escena.

Si nos fijamos en la imagen 200:



Ilustración 20: Imagen 200

Y en la imagen 201:



Ilustración 21: Imagen 201

Vemos que ambas son muy similares, pero no ha encontrado la imagen 201 cuando hemos buscado imágenes de la misma escena para la imagen 200, ¿por qué?

Vemos que es una imagen muy plana, no tenemos regiones ni ningún objeto que sea lo suficientemente característico. Esto se traduce en que no estamos consiguiendo en la imagen suficientes buenos descriptores y que no estamos detectando suficientes cambios en los gradientes.

Al fin y al cabo, estamos trabajando siempre con la detección de gradientes para obtener lo más característico de una imagen, sus palabras, y poder compararla. Entonces si nuestra imagen es plana, con una pobre detección de gradientes, no vamos a tener la suficiente información para extraer las suficientes palabras en ella, llevando la búsqueda a ambigüedades y obteniendo malos resultados tal y como vemos para la imagen 200.