VISIÓN POR COMPUTADOR

PRÁCTICA 3 – JULIO A. FRESNEDA GARCÍA – 49215154F

1. EMPAREJAMIENTO DE DESCRIPTORES [4 PUNTOS]

Mirar las imágenes en imagenesIR.rar y elegir parejas de imágenes que tengan partes de escena comunes. Haciendo uso de una máscara binaria o de las funciones extractRegion() y clickAndDraw(), seleccionar una región en la primera imagen que esté presente en la segunda imagen. Para ello solo hay que fijar los vértices de un polígono que contenga a la región.

Extraiga los puntos SIFT contenidos en la región seleccionada de la primera imagen y calcule las correspondencias con todos los puntos SIFT de la segunda imagen (ayuda: use el concepto de máscara con el parámetro mask).

Pinte las correspondencias encontrados sobre las imágenes.

Jugar con distintas parejas de imágenes, valorar las correspondencias correctas obtenidas y extraer conclusiones respecto a la utilidad de esta aproximación de recuperación de regiones/objetos de interés a partir de descriptores de una región.

Este ejercicio consiste en emparejar los descriptores de una región con los descriptores de otra imagen, para así ver si podemos detectar partes iguales en escenas comunes.

Lo primero que hacemos es obtener los puntos del polígono que forma la región que queremos, una vez tenemos los puntos creamos una máscara, y finalmente obtenemos la región.

Obtenemos los descriptores de los puntos SIFT de esa región, y los descriptores de la imagen donde queremos buscar esa región. Calculamos sus correspondencias, y las mostramos.

-Para obtener la región de la imagen, usamos getPoly(), la cual con ayuda de una máscara devuelve la región deseada.

-Para detectar y dibujar los matches entre la región y la imagen, usamos matchesRegion(), la cual llama a detectMatchesKNN(). detectMatchesKNN realiza lo siguiente:

1. Creamos un objeto bf = cv.BFMatcher (sin crossCheck pues es incompatible con knn), y obtenemos los matches usando bf.knnMatch (donde se le pasa k = 2 pues es 2NN).
2. Aplicamos el test que Lowe propuso en su paper: Guardamos los matches cuya distancia supere un cierto ratio.
3. Como nos piden que mostremos 100 aleatorios, hacemos una lista con 100 matches aleatorios
4. Dibujamos los matches entre las dos imágenes con cv.DrawMatches

No suelo incluir código en la memoria, pero creo que en esta práctica las funciones comentadas pueden complementar la explicación por si no es demasiado clara.

Por tanto, nuestra función principal es matchesRegion:

```
# Función para detectar correspondencias entre una región y una imagen distinta
def matchesRegion( img1, img2 ):

    # Usamos la función auxiliar dada en auxFunc.py para extraer una región con el ratón
    # La región la extraeremos de 'pareja1a', 'pareja2a', y 'pareja3a'
    pts = aux.extractRegion(img1)
    pts = np.asarray(pts)

    # Región seleccionada
    region = getPoly(img1,pts)

    # Detectamos los keypoints y descriptores de la región y la segunda imagen
    kp_region, desc_region = sift.detectAndCompute(region,None)
    kp_img2, desc_img2 = sift.detectAndCompute(img2,None)

    # Obtenemos los matches y los dibujamos en la imagen final
    img_matches = detectMatchesKNN(img1,img2,(kp_region,kp_img2),(desc_region,desc_img2))

    return img_matches
```

Esta función usa getPoly y detectMatchesKNN:

```
## Función que devuelve el polígono dados una imagen y unos puntos
def getPoly( img, points ):

    # Creamos una máscara vacía (negra)
    mask = np.zeros((img.shape[0], img.shape[1]))

    # Convertimos los puntos en un polígono con la función de OpenCV
    cv2.fillConvexPoly(mask, points, 1)

    # Pasamos la máscara a tipo booleano (se pinta píxel o no)
    mask = mask.astype(np.bool)

    # Finalmente con ayuda de la máscara, obtenemos sólo el polígono de la imagen
    out = np.zeros_like(img)
    out[mask] = img[mask]

    # Devolvemos el polígono
    return out


# Función para dibujar matches eligiendo según ratio test de Lowe.
# Utilizada en la práctica 2.
################         LOWE-AVERAGE-2NN             ##############################
def detectMatchesKNN( img1, img2, kps,descs, num_matches = 10, ratio = 0.8, knn = 2 ):

    # Desempaquetamos los keypoints y descriptores asociados
    (kp1, kp2) = kps
    (desc1, desc2) = descs

    # Buscamos los knn mejores matches
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(desc1,desc2, k= knn)

    # Aplicamos el test que Lowe propuso en su paper
    good_matches = []
    for m,n in matches:
        if m.distance < ratio*n.distance:
            good_matches.append([m])

    print(len(good_matches))

    # Devolvemos la imagen con los matches dibujados
    return (cv2.drawMatchesKnn(img1,kp1,img2,kp2,good_matches,None))
```

Vamos a probar estas funciones y comprobar si efectivamente es útil este método.

Empezamos por las siguientes dos imágenes:

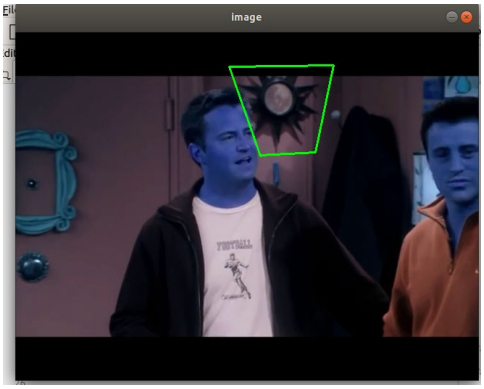
Una vez tenemos las imágenes, vamos a pasárselas a matchesRegion como argumento. La función llamará a getPoly, el cual nos pedirá que dibujemos la región:

El resultado es el siguiente.

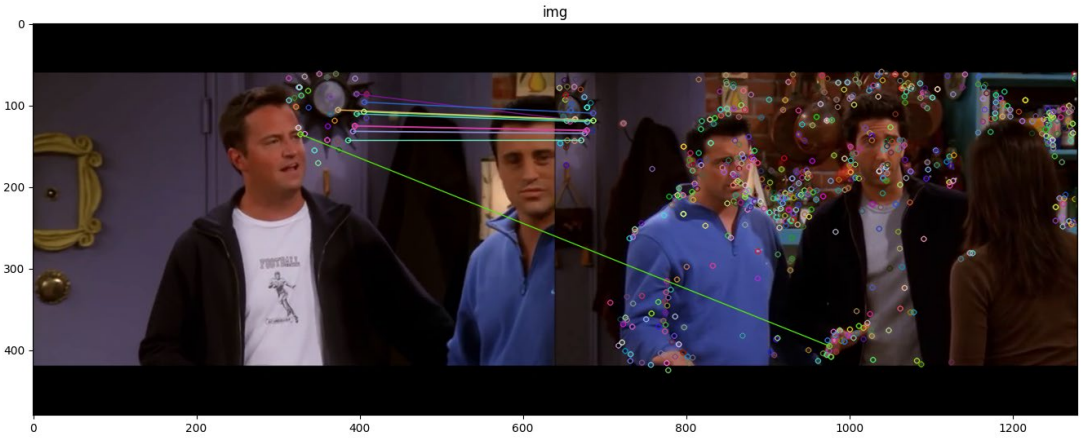
Como vemos, se encuentran correspondencias correctamente. Era fácil, ya que la camiseta tiene un dibujo muy característico.

Vamos a probar con el segundo par de imágenes:

En este caso vamos a intentar encontrar el adorno en forma de estrella de la pared en la segunda imagen:

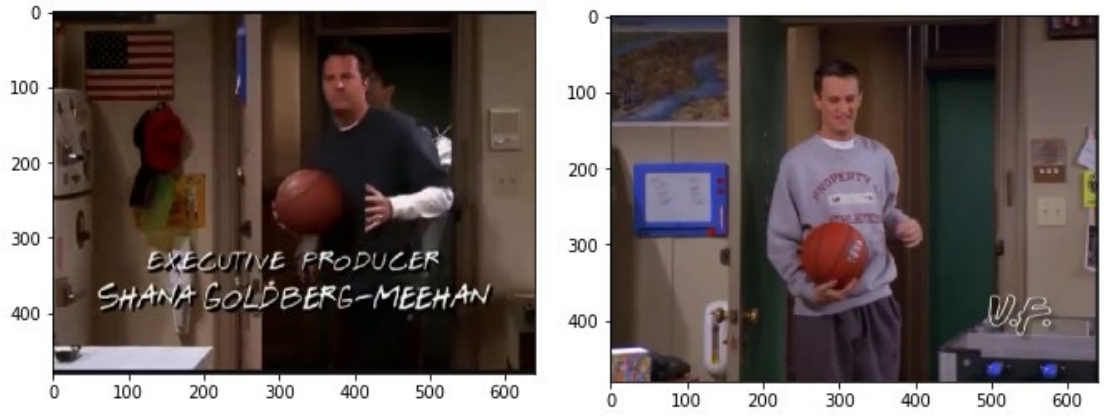


Vamos a ver qué matches obtenemos con la segunda imagen:



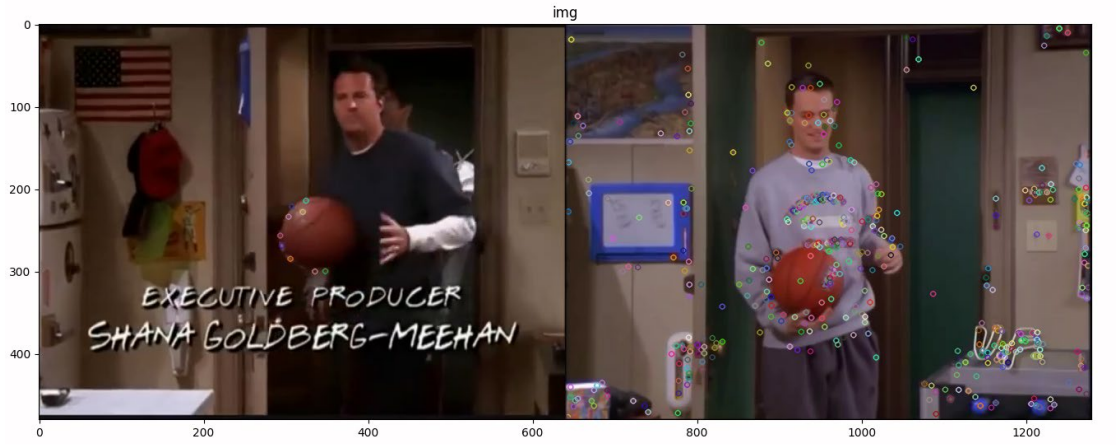
Aquí vemos que, aunque en la segunda imagen no se ve la parte izquierda, con la parte derecha es suficiente para que se hagan bastantes matches correctos.

Por último, vamos a probar con un objeto más difícil:



Vamos a intentar encontrar el balón de baloncesto de la primera imagen en la segunda imagen.

El resultado es el siguiente.



Aquí no se han encontrado correspondencias. Es normal, puesto que, aunque nosotros veamos claro la correspondencia, el algoritmo usa los gradientes, y los únicos gradientes que tiene la pelota es el borde, además de que el fondo de la pelota es distinto en cada imagen.

En conclusión, podemos decir que esta forma de buscar correspondencias es efectiva sólo cuando en la región hay patrones claros con gradientes precisos, pero no es efectiva cuando la región no tiene gradientes, como la pelota.

2. RECUPERACIÓN DE IMÁGENES [4 PUNTOS]

Implementar un modelo de índice invertido + bolsa de palabras para las imágenes dadas en imagenesIR.rar usando el vocabulario dado en kmeanscenters2000.pkl.

Verificar que el modelo construido para cada imagen permite recuperar imágenes de la misma escena cuando la comparamos al resto de imágenes de la base de datos.

Elegir dos imágenes-pregunta en las se ponga de manifiesto esto que el modelo usado es realmente muy efectivo para extraer sus semejantes y elegir otra imagen-pregunta en la que se muestre que el modelo puede realmente fallar. Para ello muestre las cinco imágenes mas semejantes de cada una de las imágenes-pregunta seleccionadas usando como medida de distancia el producto escalar normalizado de sus vectores de bolsa de palabras.

Explicar que conclusiones obtiene de este experimento.

En este ejercicio vamos a crear una clase la cual, dándole una imagen, nos devuelve las imágenes más parecidas. Para ello usaremos las siguientes funciones:

-generate_histogram() nos devuelve el histograma de una imagen

-compare_histograms() compara dos histogramas

-get_similar() obtiene los histogramas más parecidos a un histograma dado

Usaremos una clase, IndiceInvertido, ya que así sólo tendremos que generar un histograma para cada imagen al inicializar el objeto.

Esta clase tendrá implementado un índice invertido, el cual es un índice que nos dice qué imágenes contienen cada palabra. Además la clase generará una bolsa de palabras o histograma para cada una de las imágenes.

Para crear un objeto de esta clase se le piden como argumentos un diccionario y una batería de imágenes, y al inicializar, la clase crea un histograma para cada imagen con generate_histogram().

Esta clase tiene un método GetSimilarImages(), que a partir de una imagen, genera sus descriptores SIFT y su histograma, y con la función get_similar obtenemos los índices de las imágenes cuyos histogramas son los más parecidos al histograma de nuestra imagen.

Para este proceso vamos a usar el vocabulario dado en 'kmeanscenters2000.pkl'. Este archivo contiene un diccionario con 2000 palabras o centroides, las cuales usaremos para generar los histogramas.

-Función generate_histogram: Para generar un histograma de una imagen, necesitamos un diccionario de palabras y los descriptores de la imagen. Si el diccionario tiene 2000 palabras, el histograma tendrá 2000 'barras' o columnas, las cuales representan a cada palabra del diccionario. Por cada descriptor de la imagen, encontramos a qué palabra/centroide pertenece y aumentamos en 1 la 'barra' del histograma correspondiente a ese centroide. Por ejemplo, si la imagen tiene 3 descriptores pertenecientes al clúster diccionario[7], histograma[7] = 3.

-Función compare_histograms: Para comparar dos histogramas, usamos como medida de distancia el producto escalar, no hace falta normalizar ya que ya normalizamos a la hora de generar el histograma.

-Función get_similar: Para a partir de una imagen obtener las imágenes más parecidas, comparamos el histograma de esa imagen con todos los histogramas de las demás imágenes, usando compare_histograms, y seleccionamos los más cercanos.

Concretamente la función generate_histogram es la siguiente.

```
# Función para generar el histograma dados los descriptores de una imagen,
# y un diccionario de palabras
def generate_histogram( dictionary, desc ):

    # Creamos un histograma lleno de ceros
    histogram = np.zeros(len(dictionary))

    # Normalizamos tanto los descriptores como el diccionario
    norm_desc = desc
    norm_dict = dictionary
    cv2.normalize(src=desc,dst=norm_desc,norm_type=cv2.NORM_L2)
    cv2.normalize(src=dictionary,dst=norm_dict,norm_type=cv2.NORM_L2)


    # La siguiente función devuelve los índices de cada palabra del diccionario
    # que esté en los descriptores 'desc'. La he encontrado en el siguiente link:
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.vq.vq.html
    words, distances = vq(norm_desc, norm_dict)

    # Para cada uno de los índices de las palabras encontradas:
    for word in words:

        # Actualizamos el histograma sumándole 1 al índice correspondiente
        # a la palabra encontrada
        histogram[word] = histogram[word] + 1


    # Devolvemos el histograma
    return histogram
```

La función compare_histograms es la siguiente.

```
# Función que compara dos histogramas. Aplica el producto escalar normalizado,
# sim(dj, q) = <dj,q>/(||dj||x||q||), el cual sirve como medida de distancia entre histogramas
def compare_histograms( a, b ):

    # Las longitudes deben ser iguales
    if( len(a) == len(b) ):

        # Inicializamos a 0
        dq = 0
        d = 0
        q = 0

        # Para cada "barra" del histograma:
        for i in range(len(a)):

            # Aplicamos la fórmula
            dq = dq + a[i]*b[i]
            d = d + a[i]*a[i]
            q = q +  b[i]*b[i]

        # Devolvemos el resultado final
        return dq / (math.sqrt(d)*math.sqrt(q))
```

La función get_similar es la siguiente.

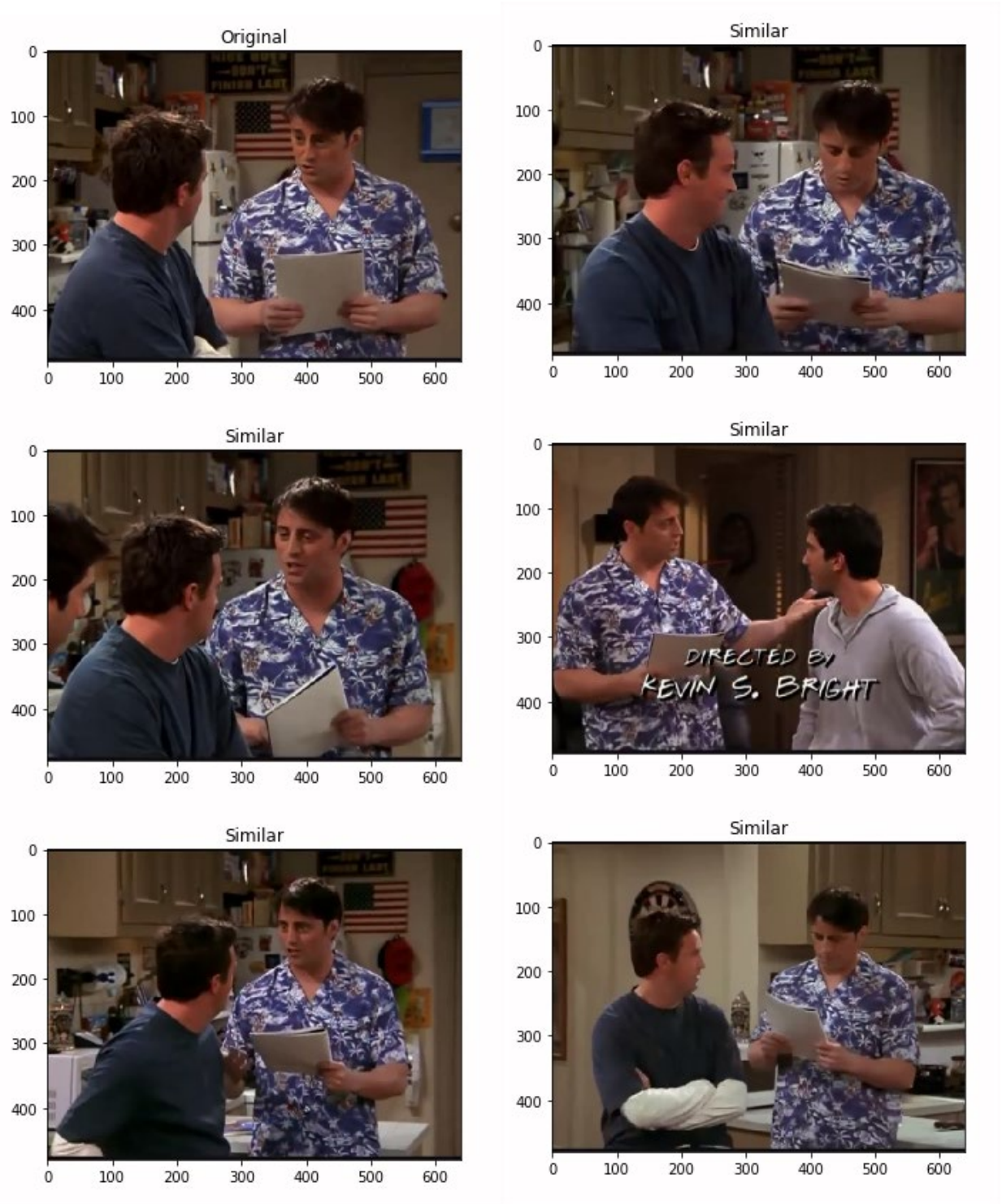
```python
# Dado un histograma, y un conjunto de histogramas, devuelve los n histogramas
# más similares al primer histograma
def get_similar( histogram, histograms, n ):

    if( n <= len(histograms)):

        # Distancias
        distances = []

        # En distancias almacenamos la distancia entre nuestro histograma y cada
        # histograma del segundo conjunto, usando la función compare_histograms
        for i in range(len(histograms)):
            distances.append(compare_histograms(histogram,histograms[i]))



        # argsort devuelve los índices de un array si estuviera ordenado:
        # https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.argsort.html
        distances_index_sorted = np.argsort(np.array(distances))

        # Como queremos los n mejores, y están ordenados de peor a mejor,
        # obtenemos los índices desde la última posición - n hasta la última posición
        res = distances_index_sorted[:len(distances_index_sorted)-n-1:-1]

        return res
```

Una vez descritas las funciones, tenemos nuestra clase IndiceInvertido, la cual nos permite, a partir de una imagen, recuperar imágenes parecidas.

```python
class IndiceInvertido:

    # Inicializamos los objetos internos
    def __init__( self, diccionario, imagenes ):
        self.imagenes = imagenes
        self.diccionario = diccionario
        self.indice = []
        self.histograms = []


        # Obtenemos los histogramas de cada imagen, usando el diccionario
        for img in self.imagenes:
            k, d = sift.detectAndCompute(img,None)
            self.histograms.append( generate_histogram(self.diccionario, d) )


        # Cargamos en el índice cada una de las palabras
        for word in self.diccionario:
            self.indice.append([])

        # Completamos el índice
        # Para cada palabra del índice:
        for w in range(len(self.indice)):
            # Para cada barra del histograma
            for h in range(len(self.histograms)):
                # Si la barra no está a 0, es que esa imagen contiene esa palabra: Esa palabra está en esa imagen
                if( self.histograms[h][w] != 0 ): self.indice[w].append(h)

    # Método que nos devuelve las <num> imágenes más parecidas a la imagen
    # cuyo índice es pasado como argumento
    def GetSimilarImages( self, index, num ):

        # Obtenemos los descriptores de la imagen
        kp, desc = sift.detectAndCompute(self.imagenes[index],None)

        # Generamos su histograma
        histogram = generate_histogram(self.diccionario, desc)

        # Buscamos los <num> histogramas más similares, obteniendo los índices
        similar_index = get_similar(histogram, self.histograms, num)

        # Obtenemos las imágenes a partir de los índices anteriores
        similar_images = []
        for index in similar_index:
            similar_images.append(self.imagenes[index])

        # Devolvemos las imágenes
        return similar_images

    def GetIndiceInvertido( self, index ):
        if( index > 0 and index < len(self.indice) ): return self.indice[index]
```

Para resolver el ejercicio, lo primero que tenemos que hacer es cargar todas nuestras imágenes en una lista. Después, cargamos el diccionario ‘kmeanscenters2000.pkl’, y creamos un objeto IndiceInvertido pasándole como argumentos tanto el diccionario como las imágenes.

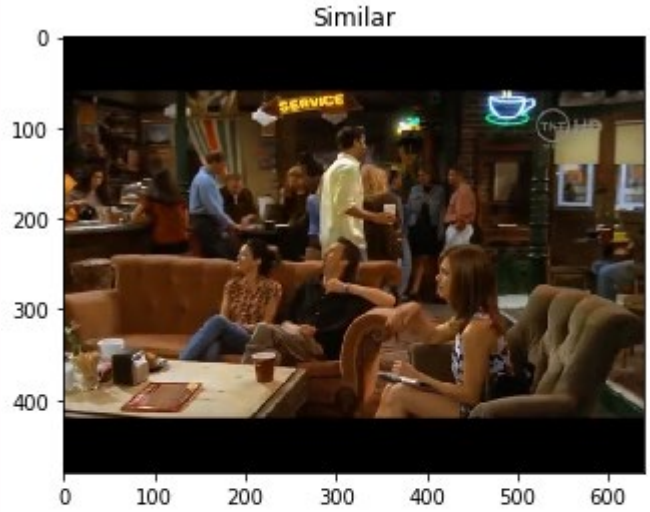
Vamos a empezar a probar con la imagen 91:

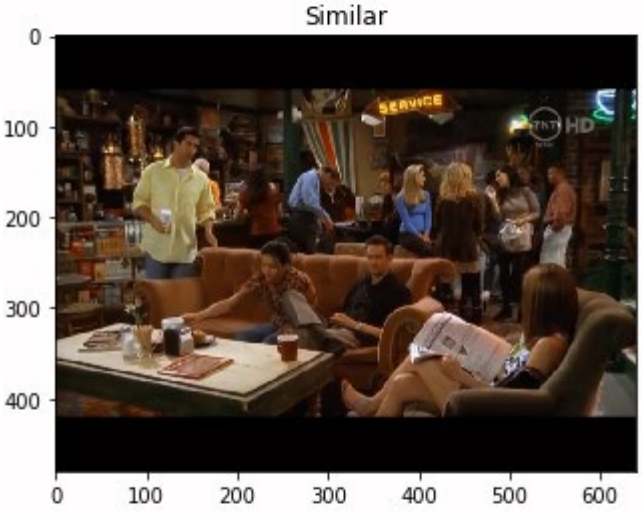


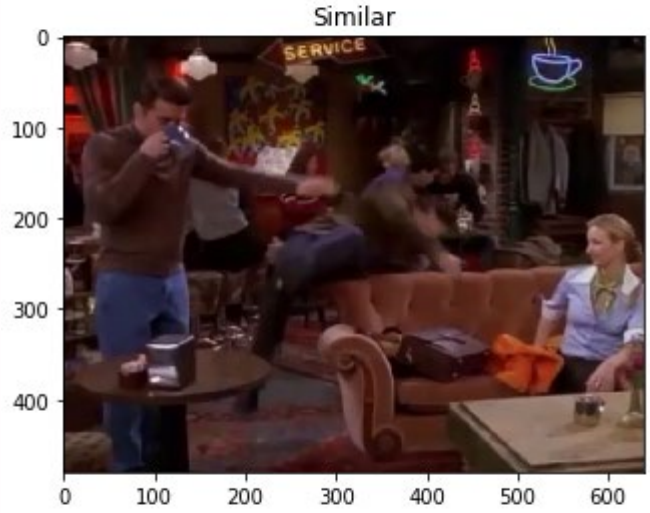
El orden de mejor a peor coincidencia es por columnas: primero las imágenes de la primera columna y después las de la segunda.

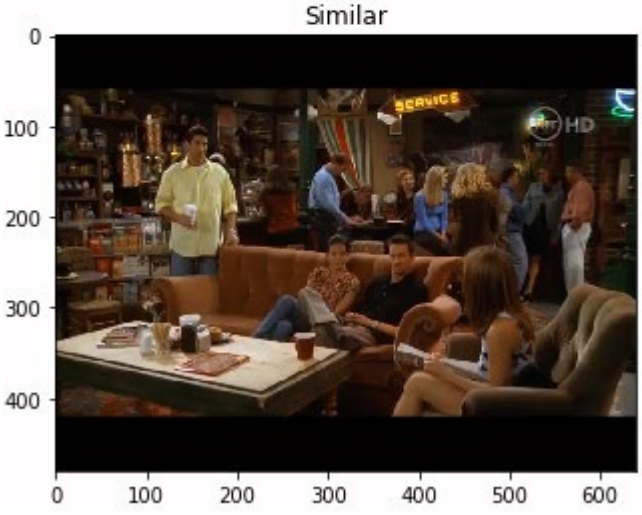
La primera imagen que obtenemos es la misma que la original, por lo que no la mostramos. Las siguientes tres imágenes son bastante parecidas, evidentemente de la misma escena. Las dos siguientes son un poco distintas, pero sigue saliendo Joey con su camisa hawaiana, por lo que podemos ver que en este ejemplo el modelo es muy efectivo, aunque hay que decir que las imágenes son muy parecidas.

Vamos a ver ahora cinco imágenes parecidas a la imagen 8.









En este caso, vemos que las 5 imágenes tienen un factor común con la original:

En todas sale el sofá. Vemos que da igual quien esté sentado, reconoce del sofá y la habitación en general perfectamente, incluso en escenas de temporadas distintas. Esto puede deberse, por ejemplo, a que el sofá tiene unos gradientes que difícilmente se darían en otra situación, como los “huecos” que hay en el respaldo del sofá. Además, favorece que tenga un color característico. En este caso vemos que el modelo es bastante efectivo.

Por último, vamos a ver cinco imágenes parecidas a la imagen 324.

En este último caso, nuestro último modelo no ha sido nada efectivo.

Ninguna de las supuestas imágenes más parecidas se parece en nada a la original. En la última salen también Ross y Monica, pero parece casualidad. Esto es porque, aunque en la original nosotros vemos claramente a Ross y Monica, es muy difícil sacar

keypoints de esa imagen, no hay nada especialmente relevante, (en el primer ejemplo tenemos la camisa hawaiana con claros patrones de Joey, y en el segundo tenemos el color característico y los bordes del sofá).

En conclusión, podemos decir que este método obtiene muy buenos resultados cuando hay patrones claros en la imagen original, como la camisa de Joey, la cual tiene dibujos bastante característicos, pero no tan buenos cuando la imagen es más genérica y sin nada especial que destaque, y con pocos bordes o gradientes pronunciados.

3. VISUALIZACIÓN DEL VOCABULARIO [3 PUNTOS]

Usando las imágenes dadas en imagenesIR.rar se han extraído 600 regiones de cada imagen de forma directa y se han re-escalado en parches de 24x24 pixeles. A partir de ellas se ha construido un vocabulario de 2.000 palabras usando k-means. Los ficheros con los datos son descriptorsAndpatches2000.pkl (descriptores de las regiones y los parches extraídos) y kmeanscenters2000.pkl (vocabulario extraído).

Elegir al menos dos palabras visuales diferentes y visualizar las regiones imagen de los 10 parches más cercanos de cada palabra visual, de forma que se muestre el contenido visual que codifican (mejor en niveles de gris). Explicar si lo que se ha obtenido es realmente lo esperado en términos de cercana visual de los parches.

En este ejercicio se nos pide elegir palabras visuales de un diccionario, es decir, centroides de un clúster, y a partir de ahí obtener los x parches cuyos descriptores de una lista de descriptores sean los más cercanos a ese clúster.

Se ha decidido hacer una clase para esta tarea, de forma que la mayoría de operaciones solo se realizan al inicializar la clase.

Para realizar este ejercicio, hay que entender bien qué contiene cada archivo.

'kmeanscenter2000.pkl' contiene la distancia (no nos interesa), las etiquetas o labels y el diccionario de 2000 palabras. Cada palabra del diccionario es un centroide de un clúster, por lo cual cada posición del diccionario corresponde a un clúster distinto. Por ejemplo, con diccionario[1980] obtenemos el centroide del clúster 1980.

Labels es una lista con 193041 etiquetas. Está relacionado con el otro archivo.

'descriptorsAndpatches2000.pkl' contiene una lista con 193041 descriptores y sus 193041 parches asociados. Estos descriptores y parches se han obtenido de las 441 imágenes de que tenemos.

Para obtener a qué clúster pertenece cada descriptor, usamos la lista labels:

Cada posición de labels corresponde a la posición del descriptor, y el contenido de esa posición de labels corresponde al clúster de ese descriptor. Por ejemplo, si queremos saber a qué clúster pertenece descriptores[4], accedemos a labels[4]. Si por ejemplo labels[4] = 1999, descriptores[4] pertenece al clúster 1999, cuyo centroide está en diccionario[1999].

Una vez descrito qué contiene cada archivo, pasamos a resolver el ejercicio.

Lo primero que hace nuestra clase es inicializar una lista clusters.

Esta lista separa los descriptores por clústeres. Pero se guarda el descriptor en sí, se guarda su índice en la lista y su distancia. Por ejemplo, si los descriptores 5, 16000 y 99000 pertenecen al clúster 1220, clusters[1220] contendría los pares (5,distancia), (16000,distancia) y (99000,distancia).

Una vez hecho esto, para cada clúster calculamos la mediana de distancia de los descriptores de un clúster respecto a su centroide. Esto no es necesario para el ejercicio, pero nos vendrá bien para elegir palabras donde se vea que los parches son parecidos (mediana baja) o no tienen mucho que ver (mediana alta).

Para obtener los índices de los parches más cercanos a un clúster en concreto, usamos getSimilarPatches(). Este método ordena por distancia los elementos de nuestro clúster de la lista de clústeres, y devuelve los índices de los <n> con menor distancia.

Para obtener el índice del clúster con mejor mediana dado un rango, usamos getClusterByMedian(). Este método ordena los clústeres por su mediana, y devuelve los índices de los <n> clústeres con menor mediana que cumplan los mínimos y máximos de mediana dados.

Todo esto se puede ver más claro en el código de la clase.

```
class ParchesCercanos:

    # Inicializamos la clase
    def __init__(self, desc, patches, labels, dictionary ):
        self.desc = desc # 193041 descriptores
        self.patches = patches # 193041 parches
        self.labels = labels # 193041 labels
        self.dictionary = dictionary # 2000 palabras

        # Vamos a crear una lista de clusters, uno por cada centroide del
        # diccionario. En cada posición habrá un par (index del desc, distancia con centroide)
        self.clusters = []

        # Inicializamos la lista
        for i in range(max(labels)[0]+1):
            self.clusters.append([])


        # Por cada uno de los descriptores
        for i in range(len(desc)):

            # Obtenemos su número de cluster
            label = labels[i][0]

            # Obtenemos la distancia euclídea entre el descriptor y el centroide de su clúster
            #https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.distance.euclidean.html
            distance = euclidean(dictionary[label], desc[i])

            # Añadimos a la lista de clusters su index en desc y su distancia con el centroide
            # Lo añadimos en la posición correspondiente al número de centroide al que pertenece
            self.clusters[label].append((i,distance))


        # Vamos a guardar una lista de medianas de distancia de los descriptores
        # de un clúster con su centroide, para ver qué descriptores
        # se asemejan de media más a sus centroides
        self.medians = []

        # Para cada clúster
        for c in self.clusters:

            # Obtenemos la distancia de cada descriptor con el centroide
            med = []
            for d in c:
                med.append(d[1])

            # Hacemos la mediana y la almacenamos
            self.medians.append(np.median(med))
```

Los métodos que usa la clase son los siguientes.

```
# Método para obtener los index de los <num> parches más similares a un centroide dado
def getSimilarPatches( self, index, num ):

    # Aquí se almacenan los índices
    patches_index = []

    # Obtenemos el cluster objetivo
    cluster = self.clusters[index]

    # Vamos a obtener las distancias
    distancias = []

    # Para cada descriptor del cluster, obtenemos su distancia y su indice
    for desc in cluster:
        distancias.append(desc[1])
        patches_index.append(desc[0])

    # Obtenemos los índices ordenados de la lista de distancias
    sort_index = np.argsort(distancias)

    # Aquí se almacena el resultado
    best_patches = []

    # Si hay más parches que el número pedido
    if( num <= len(sort_index) ):

        # <num> veces:
        for i in range(num):
            # Obtenemos el índice del parche en la posición dada en sort_index
            best_patches.append(patches_index[sort_index[i]])

    else:
        for i in len(sort_index):
            best_patches.append(patches_index[sort_index[i]])


    # Devolvemos los índices
    return best_patches



# Este método devuelve los <num> índices de cluster con mejor media entre los
# intervalos pasados como atributos.
# Este método nos va a servir para que en el ejercicio usemos clústers
# donde sus descriptores se asemejen bastante, y se vea visualmente el
# parecido
def getClusterByMedian( self, num, max_median = 1, min_median = 0 ):

    # Obtenemos los índices ordenados de las medianas
    median_index = np.argsort(self.medians)

    # Lista con los índices que pasan los umbrales dados
    index_approved = []

    # Vamos comprobando las medias de mejor a peor
    for i in range(len(median_index)):
        # Si la media cumple los umbrales, guardamos su índice
        if( self.medians[median_index[i]] <= max_median and self.medians[median_index[i]] >= min_median ):
            index_approved.append(median_index[i])


    if( num < len(index_approved) ):
        return index_approved[:num]
    else: return index_approved
```

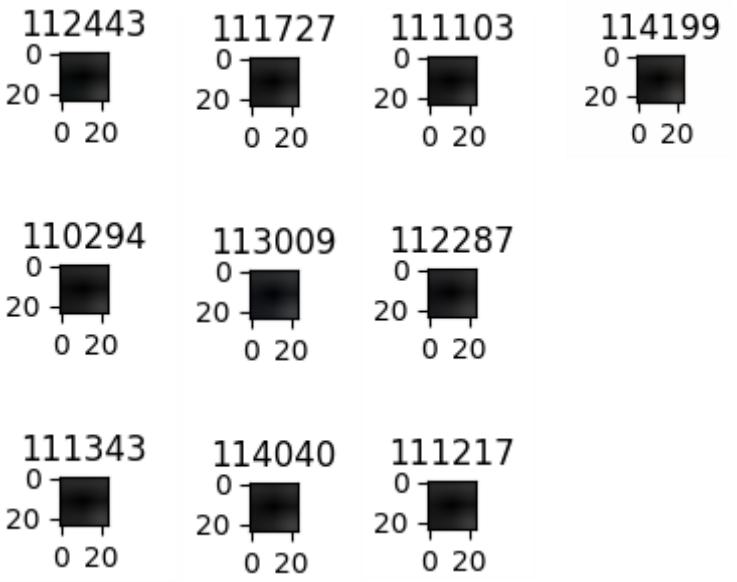
Una vez explicado cómo resolvemos el ejercicio, vamos a buscar los 10 parches más cercanos a 3 palabras visuales diferentes.

Para ello, tenemos que obtener de kmeanscenters2000.pkl nuestro diccionario y etiquetas, y de descriptorsAndpatches2000.pkl nuestros descriptores y parches.

Una vez hecho esto, creamos nuestro objeto ParchesCercanos.

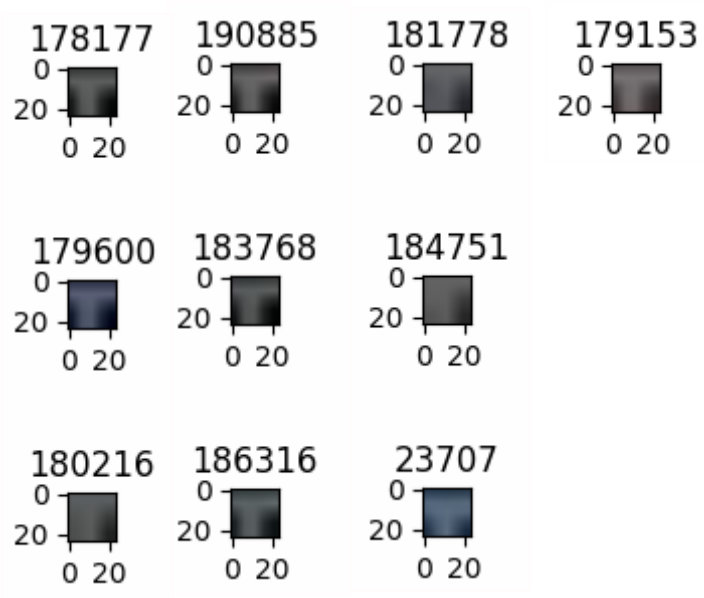
Primero vamos a buscar el mejor clúster, el clúster cuyos descriptores tengan menor distancia con él. Para ello, usamos la función getClusterByMedian, y no le ponemos mínimo de media (por defecto es 0).

Usando esta función, obtenemos el índice 850: El mejor clúster es el 850. Vamos a ver las 10 palabras que más se parecen al centroide del clúster:



Como podemos ver, los 10 parches son casi idénticos, lo cual era de esperar pues hemos cogido el mejor clúster posible.

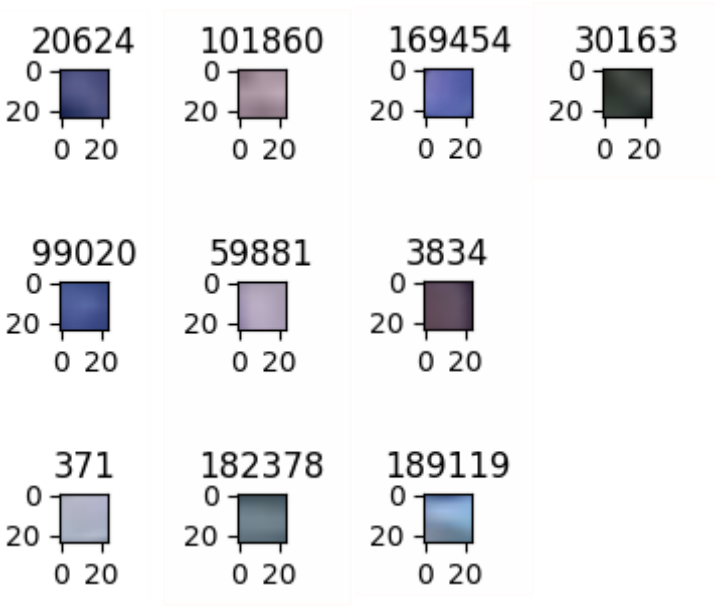
Vamos a probar ahora con un clúster algo menos bueno: Alguno cuya mediana de distancias de sus descriptores sea mayor que 0,2 por ejemplo. Si hacemos esto, obtenemos el clúster 1933. Vamos a calcular sus 10 parches más cercanos.



Este clúster es más interesante, pues, aunque vemos que no todos los parches son iguales, todos comparten unos patrones: Fondo negro y una forma de 'T' de color gris. La mediana usada es baja, por lo tanto, era de esperar que se parecieran.

Vamos ahora a obtener algún clúster algo malo, alguno cuya media sea mayor de 0.5.

Si ponemos ese mínimo como umbral, obtenemos el clúster 1974. Vamos a calcular sus 10 parches más cercanos.



Vemos que en los parches de este clúster apenas vemos parecidos. Como mucho podemos ver que las esquinas y la parte superior e inferior son algo más oscuras, pero aun así no se parecen demasiado, lo cual era esperable ya que la mediana de la distancia es bastante alta.