Visión por Computador (2017-2018)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Trabajo 3:
Indexación y recuperación de imágenes

Mª del Mar Alguacil Camarero

Índice

1. Ejercicio 1

Emparejamiento de descriptores: **aproximación en la recuperación de imágenes a través de descriptores. Extraer los puntos SIFT contenidos en la región seleccionada de la primera imagen y calcular las correspondencias con todos los puntos SIFT de la segunda imagen.**

Para esta aproximación se ha implementado una función en la que se le pasa las imágenes seleccionadas (que deben tener partes de escenas comunes).

Incialmente haciendo uso de las funciones `extractRegion` y `clickAndDraw` se selecciona una región convexa en la primera y segunda imagen que esté presente en ambas. Seguidamente calculamos las máscaras del mismo tamaño que las respectivas imágenes, poniendo a ceros todos los píxeles excepto los de la región seleccionada en cada imagen. Y con dichas imágenes calculamos las correspondencias de los puntos SIFT. Por último, dibujamos la pareja de imágenes con los emparejamientos hallados.

Correspondencias

```
def emparejamiento(filenames, leer=True):
    # Leemos y pasamos las imágenes a blanco y negro
    if leer:
        print(filenames[0])
        img1 = cv2.imread(filenames[0])
        img2 = cv2.imread(filenames[1])
    else:
        img1 = filenames[0]
        img2 = filenames[1]

    gray1 = cv2.cvtColor(img1,cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(img2,cv2.COLOR_BGR2GRAY)

    # Seleccionamos las regiones que queremos tener en cuenta
    region1 = extractRegion(gray1)
    region2 = extractRegion(gray2)

    # Calculamos las máscaras que deben ser del mismo tamaño que las respectivas
    # imágenes.
    img_nula1 = np.zeros(gray1.shape)
    mask1 = cv2.fillConvexPoly(img_nula1, np.asarray(region1), 255)

    img_nula2 = np.zeros(gray2.shape)
    mask2 = cv2.fillConvexPoly(img_nula2, np.array(region2), 255)

    # Calculamos los keypoints SIFT con dicha máscara y dibujamos las distintas
    # imágenes con las correspondencias obtenidas.
```

```
emparejarSift([img1, img2], dibujar=True, leer=False, N=100, M=-1, f=0.65,
                mask1=np.array(mask1, dtype=np.uint8),
                mask2=np.array(mask2, dtype=np.uint8), ej1=False)
```

A continuación se muestran algunos ejemplos:

Figura 1.1: Correspondencias imágenes 420 y 422.

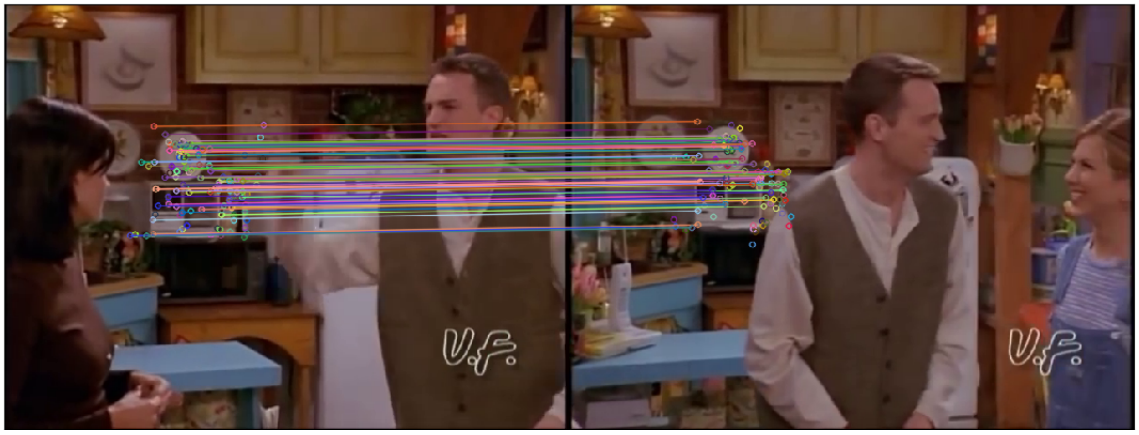

Figura 1.2: Correspondencias imágenes 156 y 157.

Figura 1.3: Correspondencias imágenes 0 y 2.

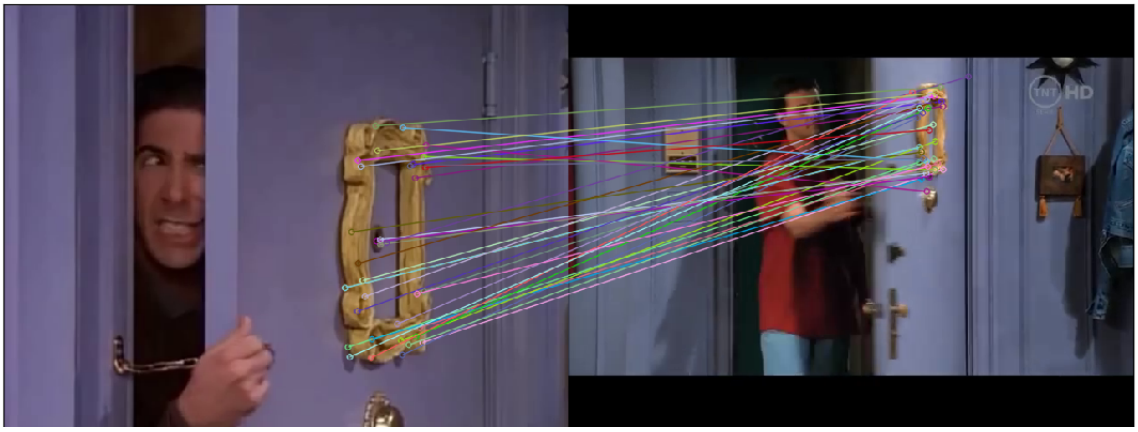

Figura 1.4: Correspondencias imágenes 36 y 50.

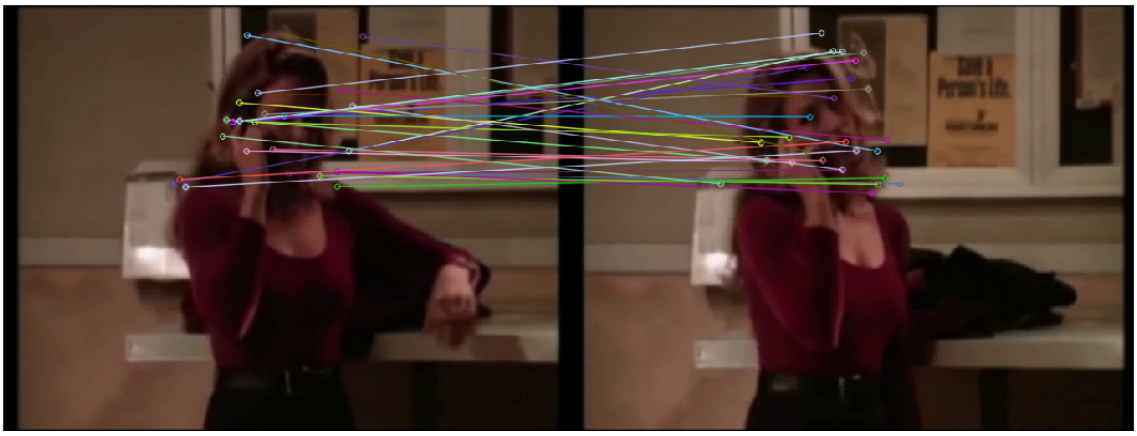

Figura 1.5: Correspondencias imágenes 165 y 166.

Figura 1.6: Correspondencias imágenes 270 y 271.

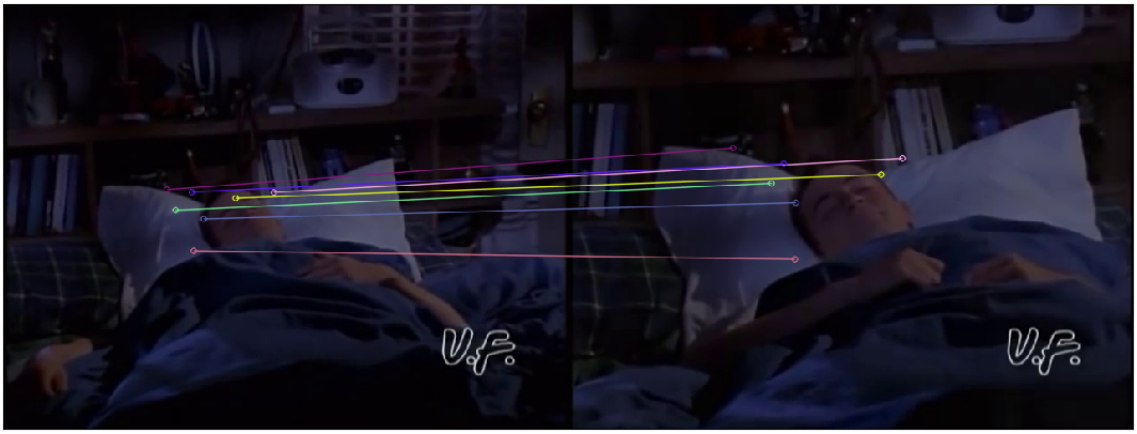

Figura 1.7: Correspondencias imágenes 137 y 138.

Los algoritmos SIFT proporcionan puntos de interés invariantes a escala, rotación y en cierta parte a iluminación. Dichas soluciones aportan gran cantidad de información por cada una de las imágenes procesadas. Sin embargo, debido a cambios de intensidad luminosa, bajo contrastre, seleccionar zonas planas tales como la cara, etc, se producen bastantes *outliers*, como se puede apreciar en las figuras 1.5, 1.6 y 1.7. Además también pueden producirse malos emparejamientos debido a la ambigüedad de ciertos puntos, como es evidente en el 1.4.

SIFT es una técnica de coincidencia extraordinariamente robusta, y es rápida y eficiente, siendo muy útil a la hora de resolver problemas geométricos. Pero para la recuperación de imágenes, es muy importante, a parte de las zonas relevantes que nos puede aportar esta técnica, tener en cuenta las zonas planas. Necesitamos un muestreo denso, información redundante para afirmar una y otra vez.

2. Ejercicio 2

Visualización del vocabulario: **Usando las imágenes dadas se han extraído regiones de cada imagen y se ha construido un vocabulario palabras usando k-means. Se han extraido de forma directa las regiones imágenes asociadas y se han re-escalado. Elegir al menos dos palabras visuales diferentes y visualizar las regiones imagen de los parches más cercanos de cada palabra visual, de forma que se muestre el contenido visual que codifican.**

Tras seleccionar nosotros 2 palabras visuales distintas, el programa se encargada de leer los ficheros (en este caso, `descriptorsAndpatches.pkl` y `kmeanscenters5000.pkl`) que contienen los distintos descriptores, parches y palabras asociadas, respectivamente, con la ayuda de las funciones `loadAux` y `loadDictionary`, transformando las regiones imagen

de los parches a blanco y negro para una mejor visualización. Creamos una matriz con las distintas etiquetas que nos indican el clusters al que pertenecen, y con la posición del descriptor asociado.
Normalizamos los descriptores dividiendo por la norma de este, para que tengan norma cuadrática igual a uno. Seguidamente calculamos las distancias de cada descriptor a la palabra, ordenándolas para poder extraer más fácilmente los N parches más cercanos. Y por último, visualizamos dichos parches asociados a cada palabra visual para poder mostrar el contenido visual que codifican.

Parches

```
def visVocabulario(vocabulario, regiones, p1, p2, N=15):
    # Leemos los ficheros
    descriptores, patches = loadAux(regiones, True)
    precision, etiquetas, palabras = loadDictionary(vocabulario)

    # Pasamos los parches a blanco y negro
    parches = []
    for patch in patches:
        parches.append(cv2.cvtColor(patch,cv2.COLOR_BGR2GRAY))

    # Matriz con las etiquetas(cluster al que pertenece) y la posición de los descriptores
    tam = len(parches)
    labels = np.squeeze(np.asarray((etiquetas)))[0:tam]
    labels = np.array([labels, list(range(tam))]).T

    labelsP1 = labels[np.where(labels[:,0] == p1)][:,1]
    labelsP2 = labels[np.where(labels[:,0] == p2)][:,1]

    # Normalizamos los descriptores para que tengan norma cuadrática igual a uno
    norma = np.sqrt((descriptores**2).sum(axis=1))
    norma = np.matrix([norma]*descriptores.shape[1]).T

    descriptores = np.array(np.divide(descriptores, norma))

    # Vector distancias
    distanciasP1 = np.matrix(palabras[p1]) * descriptores[labelsP1].T
    distanciasP1 = np.squeeze(np.asarray(distanciasP1))
    distanciasP2 = np.matrix(palabras[p2]) * descriptores[labelsP2].T
    distanciasP2 = np.squeeze(np.asarray(distanciasP2))

    # Matriz que contiene las distancias calculadas y la posición asociada
    distPosP1 = np.array([distanciasP1, labelsP1]).T
    distPosP2 = np.array([distanciasP2, labelsP2]).T
```

```
    # Ordenamos las distancias
    distPosP1 = distPosP1[distPosP1[:, 0].argsort()]
    distPosP1 = distPosP1[::-1]
    distPosP2 = distPosP2[distPosP2[:, 0].argsort()]
    distPosP2 = distPosP2[::-1]

    # Extraemos los índices de los N parches más cercanos
    S = min(N, distPosP1.shape[0])
    indices1 = np.array(distPosP1[0:S,1], dtype = np.int32)
    T = min(N, distPosP2.shape[0])
    indices2 = np.array(distPosP2[0:T,1], dtype = np.int32)

    # Convertimos la lista de parches en un array
    parches = np.array(parches)

    # Visualizar las regiones imagen de los N parches más cercanos de cada palabra visual
    visualization(parches[indices1], ['']*S, 3, S/3, color = False)
    visualization(parches[indices2], ['']*T, 3, T/3, color = False)
```

Como suplemento, para una representación de parches más coherente, se ha implementado una función para seleccionar las *num_palabras* con menor dispersión. Para ello se ha calculado la varianza de todos los clusters y seleccionado los que están más agrupados.

Menor dispersión

```
# Varianza de la palabra deseada
def varianza(descriptores, palabra):
    varianzas = np.squeeze(np.asarray(palabra*descriptores.T))

    return max(varianzas)

# Seleccionamos las num_palabras con menor dispersión
def varianzas(vocabulario, regiones, num_palabras=2):
    # Leemos los ficheros
    descriptores, parches = loadAux(regiones, True)
    precision, etiquetas, palabras = loadDictionary(vocabulario)

    # Normalizamos los descriptores para que tengan norma cuadrática igual a uno
    norma = np.sqrt((descriptores**2).sum(axis=1))
    norma = np.matrix([norma]*descriptores.shape[1]).T

    descriptores = np.array(np.divide(descriptores, norma))

    # Matriz con las etiquetas, los descriptores y los parches
    tam = len(parches)
```

```
    labels = np.squeeze(np.asarray((etiquetas)))[0:tam]
    labels = np.array([labels, list(range(tam))]).T

    # Varianzas de los distintos clusters
    N = palabras.shape[0]
    var = [None]*N
    for p in range(N):
        labelsP = labels[np.where(labels[:,0] == p)][:,1]
        var[p] = varianza(descriptores[labelsP], np.matrix(palabras[p]))
    var = np.array([var, list(range(N))]).T
    var = var[var[:, 0].argsort()]
    var = var[::-1]

    # Devolvemos los índices de las palabras
    return np.array(var[:num_palabras, 1], dtype=int)
```

A continuación en las figuras 2.1 y 2.2 se muestran los 15 parches correspondientes a las dos palabras con menor dispersión, seleccionadas con la ayuda de la función antes explicada. Y además, en las figuras 2.3 y 3.1 los 15 parches de dos palabras aleatoriamente elegidas por mí.

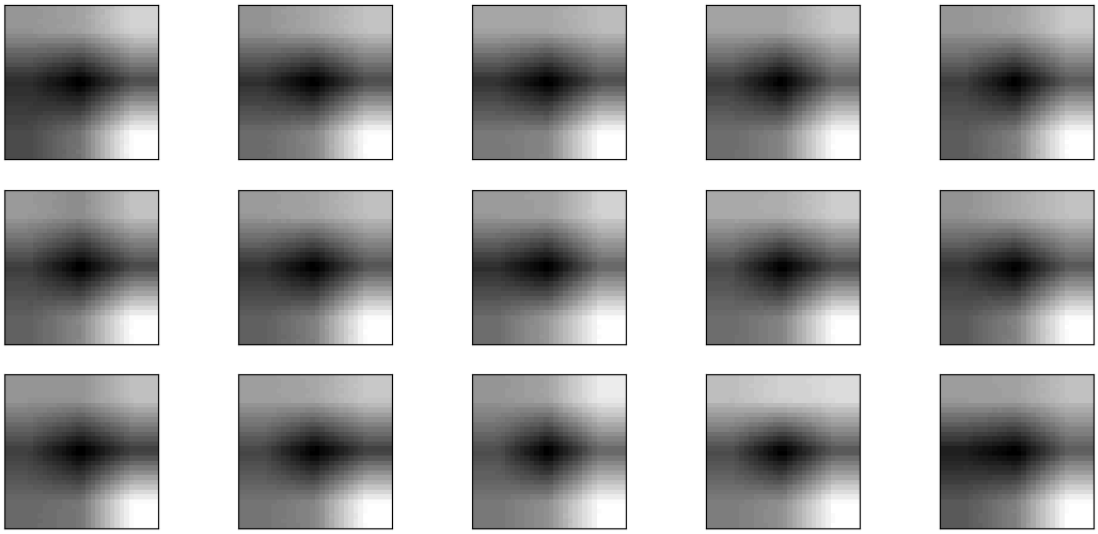

Figura 2.1: 15 parches más cercanos de la palabra 1578.

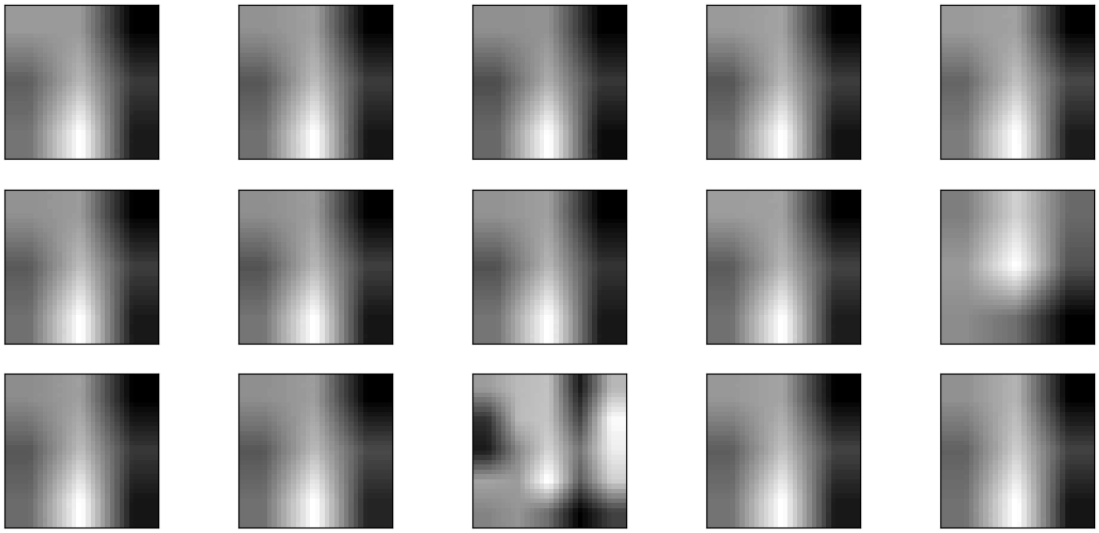

Figura 2.2: 15 parches más cercanos de la palabra 4247.

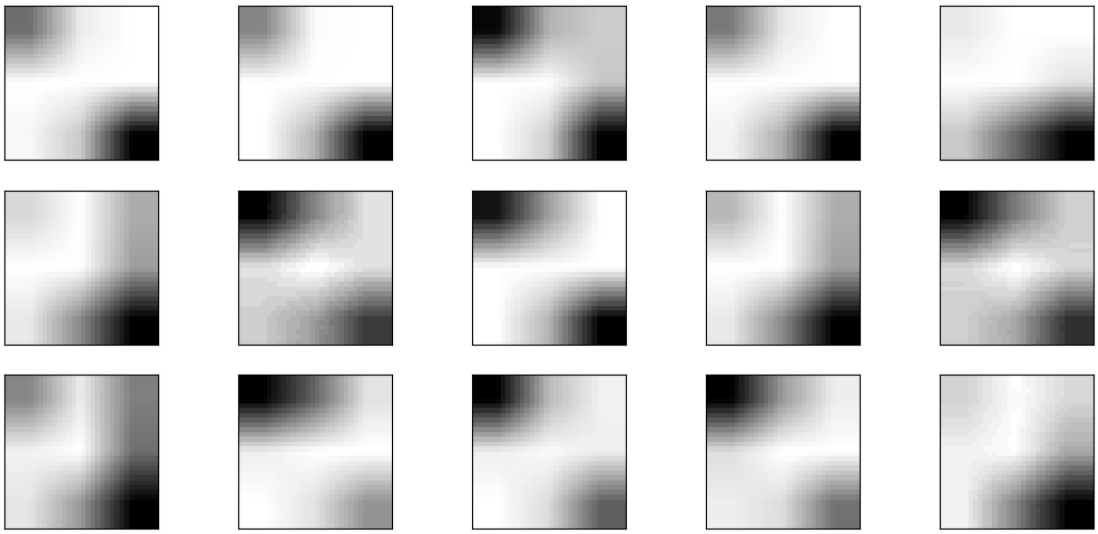

Figura 2.3: 15 parches más cercanos de la palabra 1662.

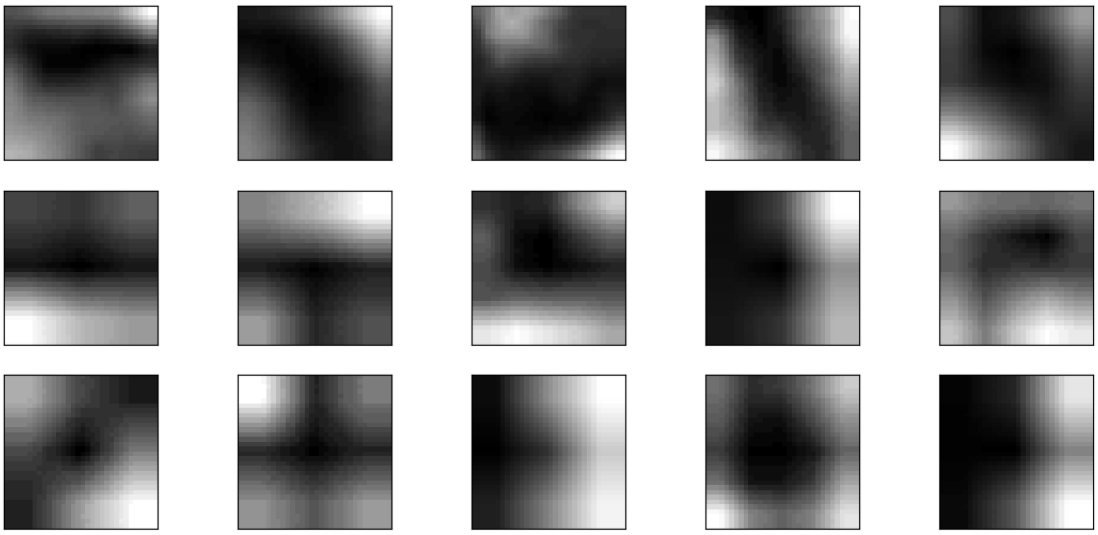

Figura 2.4: 15 parches más cercanos de la palabra 500.

3. Ejercicio 3

Recuperación de imágenes: **Implementar un modelo de índice invertido + bolsa de palabras para las imágenes dadas. 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 que se ponga de manifiesto 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 más 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.**

La idea de la recuperación de imágenes es votar palabras, entendiendo, en esta caso, por palabra visual la media de un conjunto de valores, descriptores. Es decir, elegimos un representante para cada conjunto de puntos.

Para realizar esta tarea nos ayudaremos de un modelo de índice invertido y de la bolsa de palabras. Para la construcción de estos inicialmente se leerá todas las imágenes que deseamos considerar en nuestra base de datos, y las palabras que formarán el vocabulario. Tras dichas lecturas procedemos a extraer los descriptores de las distintas imágenes. Seguidamente creamos la tabla de índice invertido de tamaño igual al número de palabras, que inicialmente estará vacía, e iremos rellenando con las imágenes en las que aparece cada palabra. Para ello, normalizamos los distintos descriptores anteriormente hallados,

calculamos las distancias e introducimos en el fichero las imágenes en las que aparece cada palabra (ayudándonos de un *umbral* prefijado, para evitar los posibles errores y obtener una mayor eficiencia del algoritmo).
A continuación contruimos los histogramas de las distintas imágenes calculando el número de veces que aparece cada palabra, y lo normalizamos para la comparación que haremos posteriormente en la recuperación de imágenes.
Por último, comparamos los distintos histogramas para determinar las N imágenes más semejantes a las imágenes-pregunta introducidas usando como medida de distancia el producto escalar normalizado de sus vectores de bolsa de palabras.
Y finalmente, representamos cada imagen-pregunta con sus N más semejantes y sus histogramas asociados.

Recuperación de imágenes

```
def ficheroInvertido(filenames, vocabulario, umbral = 1.7):
    # Leemos todas la imagenes
    imagenes = []
    for file in filenames:
        imagenes.append(cv2.imread(file, 0))

    # Leemos el vocabulario
    precision, etiquetas, palabras = loadDictionary(vocabulario)

    # Inicializamos SIFT
    sift = cv2.xfeatures2d.SIFT_create()

    # Detectamos los keypoint y extraemos los descriptores de las distintas imágenes
    desc = []
    for img in imagenes:
        kpts, dcts = sift.detectAndCompute(img, None)
        desc.append(dcts)

    # Creamos una tabla de indice invertido de tamaño igual al número de palabras
    fichero = [np.copy([])]*palabras.shape[0]
    for j, descriptor in enumerate(desc):
        # Normalizamos los descriptores
        max_desc = np.max(descriptor)
        descriptor = np.matrix(descriptor/max_desc)

        # Matriz de distancias
        distancia = palabras*descriptor.T

        # Seleccionamos la distancia mayor
        # Máximo de cada fila (posición)
        maximos = np.argmax(distancia, axis=0)
```

```
            for i in range(maximos.shape[1]):
                maximo = maximos[0,i]
                if distancia[maximo,i] > umbral:
                    fichero[maximo] = np.append(fichero[maximo], j)

        # BOLSA DE PALABRAS
        N = len(imagenes)
        # Incializamos los histogramas con 0's
        histogramas = np.zeros((N, len(fichero)))
        for j, palabra in enumerate(fichero):
            if len(palabra)!=0: # Si existe alguna imagen con esa palabra
                palabra = np.array(palabra)
                for i in range(N):
                    histogramas[i,j] = len(np.where(palabra==i)[0])

        # Normalizamos el histograma
        for i in range(N):
            maxi = max(histogramas[i])
            histogramas[i] /= maxi


        return histogramas


    def recuperacion(filenames, vocabulario, indice, N=5):
        histogramas = ficheroInvertido(filenames, vocabulario)
        # Comparamos histogramas y elegimos las N imagenes mas parecidas
        errores = []
        h = histogramas[indice]
        for i,g in enumerate(histogramas):
            if i!=indice:
                e = np.dot(g, h)/(sqrt(np.dot(g, g))*sqrt(np.dot(h, h)) )
                errores.append((e,i))

        errores.sort(key=lambda x: x[0], reverse=True)

        mejores = list(map(lambda x: x[1], errores[0:N]))

        hist = [histogramas[indice]]
        img = [cv2.imread(filenames[indice])]
        for i in range(N):
            ind = mejores[i]
            img.append( cv2.imread(filenames[ind]) )
```

```
    hist.append( histogramas[ind] )

# Visualizamos las distintas imagenes y los histogramas asociados
visualization(img, ['']*(N+1), 2, (N+1)/2, color = True)
visualization(hist, ['']*(N+1), 2, (N+1)/2, plot=True)
```

Figura 3.1: 5 imágenes más semejantes a la 2.

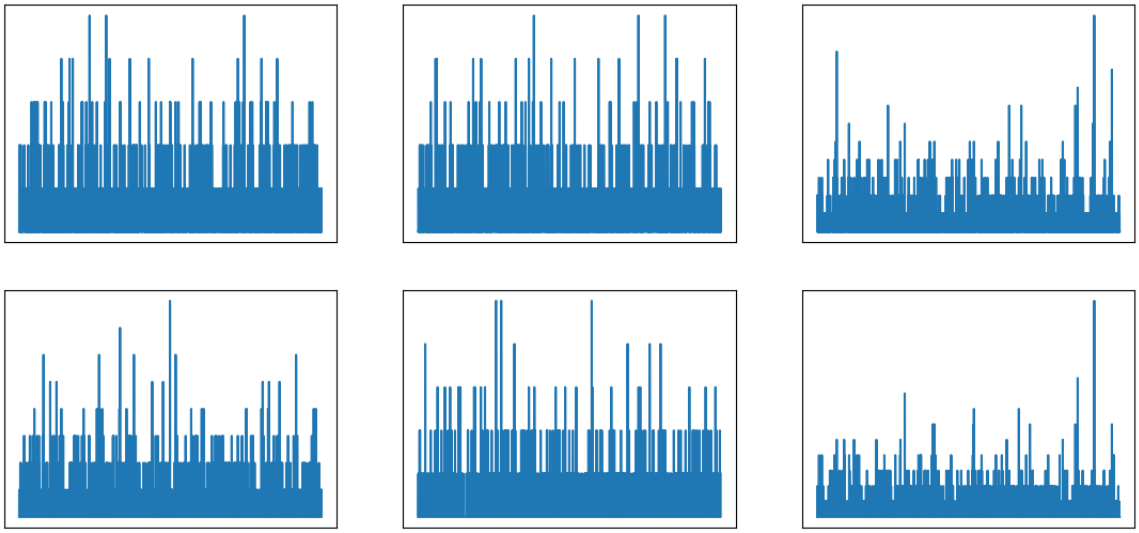

Figura 3.2: Histograma de la imagen 2 y sus 5 más semejantes.

Figura 3.3: 5 imágenes más semejantes a la 377.

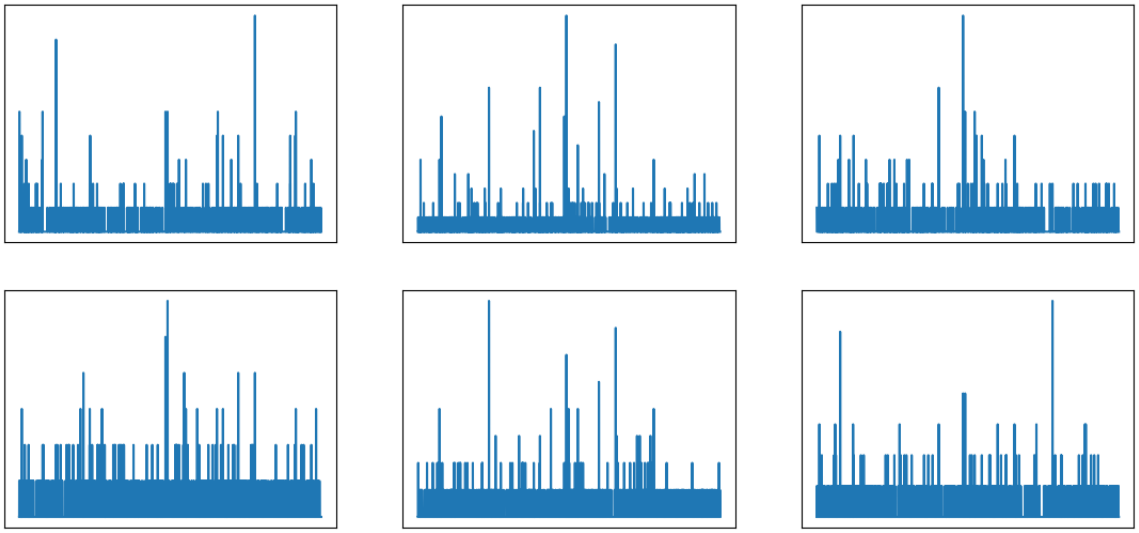

Figura 3.4: Histograma de la imagen 377 y sus 5 más semejantes.

Figura 3.5: 5 imágenes más semejantes a la 52.

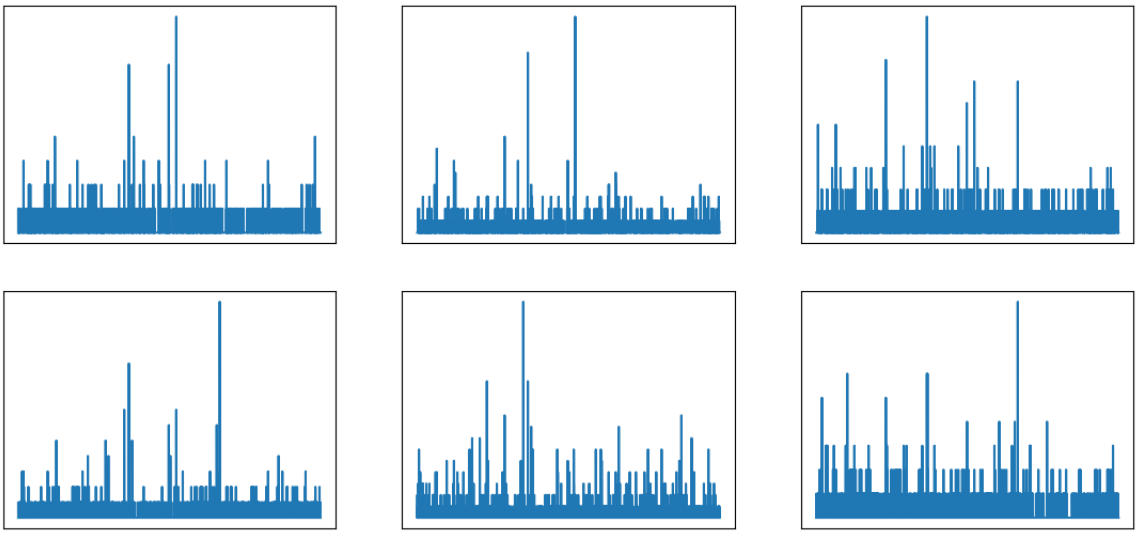

Figura 3.6: Histograma de la imagen 52 y sus 5 más semejantes.

Figura 3.7: 5 imágenes más semejantes a la 62.

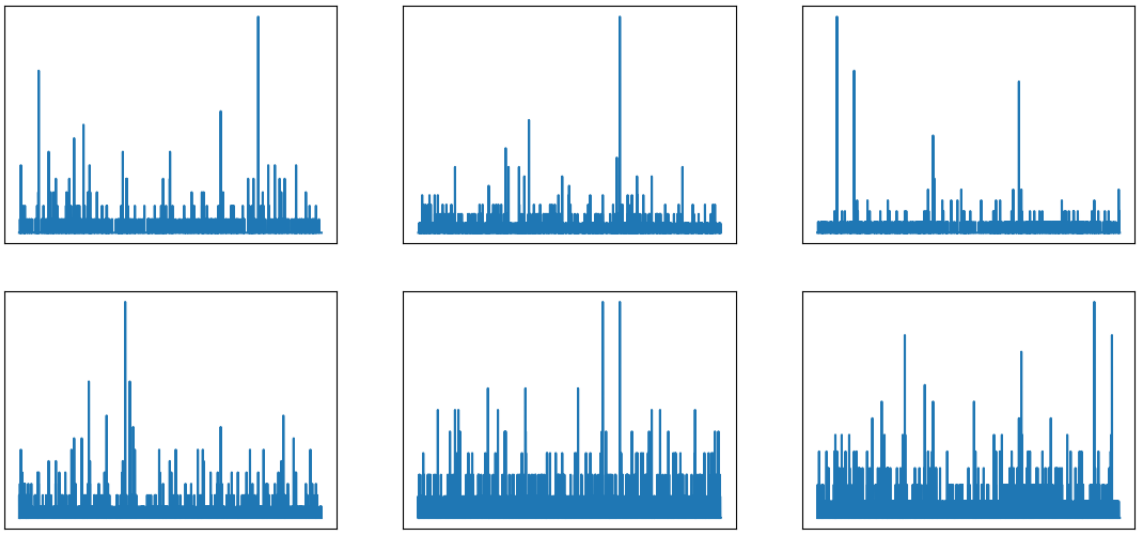

Figura 3.8: Histograma de la imagen 62 y sus 5 más semejantes.

Figura 3.9: 5 imágenes más semejantes a la 36.

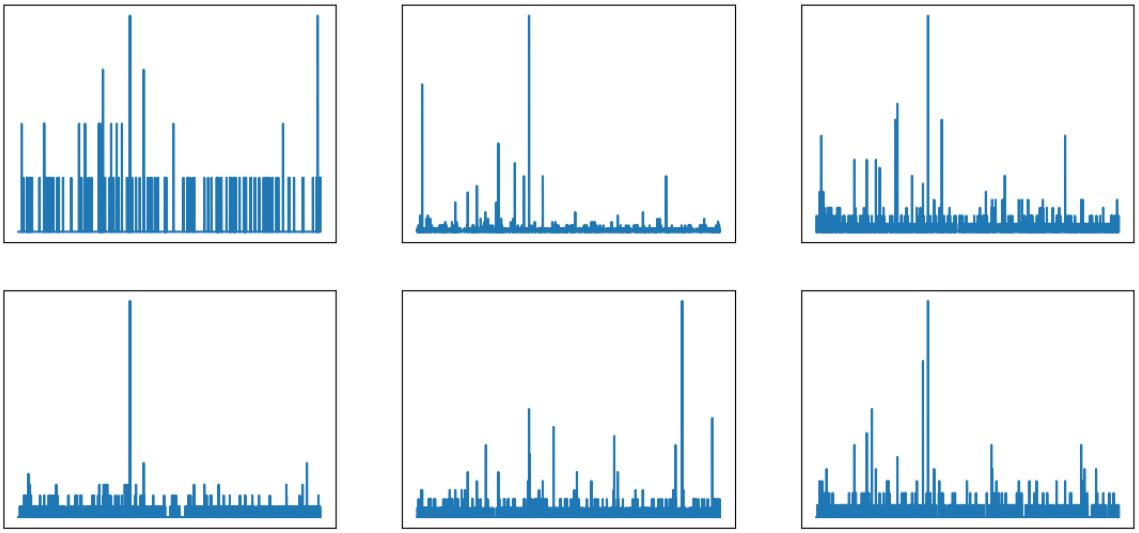

Figura 3.10: Histograma de la imagen 36 y sus 5 más semejantes.

La recuperación de las imágenes emplea como medida de distancia el producto escalar normalizado de sus vectores de bolsa de palabras. Y puede producirse que imágenes muy diferentes tengas histogramas muy parecidos debido al vocabulario empleado, como se puede observar claramente en la figura 3.7 y 3.8. También puede producirse errores debido a que la imagen tenga una diferencia considerable de palabras con respecto a otra/s que la contienen, como ocurre en el caso de la imagen representada en la figura 3.9.

En consecuencia, se pone de manifiesto que tanto la elección del vocabulario como su tamaño es muy influyente. Además de que es importante muestrear tanto en zonas planas como en las relevantes, como se ha explicado en el ejercicio 1.