



ugr | Universidad
de **Granada**

Grado en Ingeniería Informática. Cuarto.

Práctica 3.

Nombre de la asignatura:

Visión por Computador. Jueves de 10:30 a 12:30.

Realizado por:

Néstor Rodríguez Vico. DNI: 75573052C.

email: nrv23@correo.ugr.es



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS
INFORMÁTICA Y DE TELECOMUNICACIÓN.

Granada, 22 de diciembre de 2017.

Índice

1 Aclaración.	3
2 Emparejamiento de descriptores.	3
3 Visualización del vocabulario.	7
4 Recuperación de imágenes.	12

1. Aclaración.

Se ha usado la función de visualizar imágenes de las prácticas anteriores.

2. Emparejamiento de descriptores.

Para este ejercicio he tenido que introducir la región a mano ya que no he conseguido usar la funciones correctamente en macOS. Voy a hacer los emparejamientos de 128-130, 143-145 y 229-232. He usado los siguientes puntos como regiones de la primera imagen de cada pareja:

```
1 pts_128 = [(425, 253), (505, 274), (580, 256), (480, 245)]  
2 pts_143 = [(348, 42), (348, 129), (452, 129), (452, 42)]  
3 pts_229 = [(393, 111), (393, 147), (408, 147), (408, 112)]
```

La función implementada es bastante similar a lo que hemos hecho en la práctica anterior, con la única diferencia de que no podemos buscar *keypoints* en toda la primera imagen, sino sólo en la región seleccionada. La función es la siguiente:

```
1 def sift_descriptor(img1, img2, points, k=0.7):  
2     descriptor = cv2.xfeatures2d.SIFT_create()  
3  
4     mask = np.zeros(shape=(np.shape(img1)[0], np.shape(img1)[1]))  
5     cv2.fillConvexPoly(mask, np.array(points, dtype=np.int32), color=1)  
6  
7     kp1, desc1 = descriptor.detectAndCompute(image=img1, mask=np.array(mask, dtype=np.  
8         uint8))  
9     kp2, desc2 = descriptor.detectAndCompute(image=img2, mask=None)  
10  
11    bf_matcher = cv2.BFMatcher(normType=cv2.NORM_L2, crossCheck=False)  
12    all_matches = bf_matcher.knnMatch(desc1, desc2, 2)  
13    matches = [m[0] for m in all_matches if m[0].distance < m[1].distance * k]  
14  
15    full = cv2.drawMatches(img1=img1, keypoints1=kp1, img2=img2, keypoints2=kp2,  
16        matches1to2=matches, outImg=None, flags=2)  
17  
18    return full
```

La idea es tener una máscara del mismo tamaño que la imagen (línea 4) e indicar que región queremos usar para encontrar *keypoints*. Para ello hacemos uso de la función *fillConvexPoly*, la cuál nos permite, dado una lista (los puntos mencionados al principio de este apartado) y una imagen, llenar el polígono asociado a los puntos de un color indicado dentro de la imagen. Como bien sabemos, *OpenCV* interpreta las máscaras como 0 y distinto de 0, por lo tanto con indicar que el polígono sea de “color” 1 es suficiente (línea 5). Una vez tenemos la máscara, sólo debemos pasársela a la función *detectAndCompute* y el resto es igual que en la práctica anterior. El valor del parámetro *flags* de la función *drawMatches* lo usamos para pintar sólo los *keypoints* para los que los que se han una correspondencia.

Como método de extracción de correspondencias he usado *kNN*, usando un umbral de 0.7 por defecto. Veamos los resultados:

Images: 128 - 130.



Images: 143 - 145.



Images: 229 - 232.



Como podemos ver, obtenemos unos resultados bastante buenos. Pero, vamos a jugar un poco con ese umbral. Usemos ahora 0.9:

Images: 128 - 130.



Images: 143 - 145.



Images: 229 - 232.



Como podemos ver, el resultado es bueno pero ya no tanto. Viendo esto nos debemos plantear que si esta técnica es realmente buena para la recuperación de imágenes. A priori parece una buena técnica. Pero debemos pensar que estamos haciendo realmente. Todo el proceso está basado en los *keypoints* y descriptores que extraemos de las imágenes. Al usar el descriptor SIFT, estos son extraídos usando los gradientes y por lo tanto nuestros resultados están ligados a la calidad de los mismos. Si estos gradientes no son capaces de extraer la información que realmente es relevante de una región o imagen determinada, obtendremos unos resultados incorrectos, ya que todo el proceso se basa en los gradientes.

Vemos ahora un ejemplo en el que, a pesar de utilizar un valor de $k = 0.7$ los resultados no son para nada buenos:

Images: 128 - 216.



La región de consulta indicada es la misma que en las imágenes anteriores, la zona del futbolín. Pero en este caso no se encuentra ningún *match*, a pesar de aparecer el mismo futbolín en la imagen. Debemos tener en cuenta que las transformaciones que ha sufrido

el futbolín en la imagen 216 con respecto a las 128 son bastante fuertes, y por eso el resultado no es bueno.

3. Visualización del vocabulario.

Para este ejercicio voy a usar los ficheros proporcionados para ello. Veamos primero el código y sobre el voy a explicar el proceso seguido:

```

1 def plot_clusters(index):
2     descriptors, patches = auxFunc.loadAux("./descriptorsAndpatches.pkl", True)
3     accuracy, labels, dictionary = auxFunc.loadDictionary("./kmeanscenters" + str(index)
4                 ) + ".pkl")
5
6     labels_enumerated = list(enumerate(labels.flatten()))
7     dict_clusters = collections.defaultdict(list)
8     for i, c in labels_enumerated:
9         dict_clusters[c].append(i)
10
11    selected_descriptors = []
12    vars = []
13    for i in range(len(dictionary)):
14        distance = np.sqrt(np.sum((descriptors[dict_clusters[i]] - dictionary[i]) ** 2,
15                                   axis=1))
16        my_index = np.argsort(distance)
17        twenty_selected = my_index[0:20] if len(my_index) >= 20 else my_index
18        my_descriptors = np.array(dict_clusters[i])[twenty_selected]
19        selected_descriptors.append(my_descriptors)
20        var = np.mean(np.var(descriptors[my_descriptors], axis=0))
21        vars.append(var)
22
23    vars_sorted = np.argsort(vars)
24    selected_clusters = vars_sorted[0:5]
25    final_descriptors = [selected_descriptors[e] for e in selected_clusters]
26    print(index, selected_clusters)
27
28    images = [np.array(patches)[d] for d in final_descriptors]
29    names = [[str(e) for e in d] for d in final_descriptors]
30    [plot_images(img, n, 4, 5, "2_" + str(index) + "_Cluster_" + str(c)) for img, n, c
31     in zip(images, names, selected_clusters)]
```

Para este ejercicio lo que vamos a hacer es ver si los descriptores asociados a un cluster son realmente similares. Que esto se cumpla es la idea que subyace bajo la idea de diccionario. Es decir, tenemos una palabra (centroide de nuestro algoritmo) que representa a una serie de descriptores. Por lo tanto, primero leemos la información necesaria (líneas 2 y 3) y construimos un diccionario que contiene como llave el índice del cluster y para cada clave los descriptores asociados a dicho cluster. Esto lo sabemos gracias al vector *labels*, el cual contiene precisamente la información necesaria. Este proceso lo podemos ver en las líneas 5-8. Una vez tenemos una estructura que nos permite acceder rápidamente a los elementos de cada cluster, vamos a procesar cada cluster (línea 12). Dentro de cada cluster puede que haya descriptores que estén bastante alejados, es decir, que el centroide no represente bien a dicho descriptor. Para que estos descriptores no influyan en nuestro calculo, vamos a quedarnos con los que están más representados por el centroide, los cuáles son los más cercanos al mismo. Así que calculamos la distancia de todos los descriptores al centroide del cluster (línea 13), los ordenamos (línea 14) de menor a mayor distancia y cogemos los 20 primeros, si los hubiese (línea 20). De esta

forma, hemos reducido los clusters a un tamaño máximo de 20 descriptores, los cuáles son los que mejor representa el centroide.

Ahora mismo tenemos una serie de palabras (clusters) con los mejores descriptores. Teóricamente yo podríamos visualizar los resultados, pero, ¿de que cluster lo hacemos? Lo que vamos a ver es que clusteres tienen menor varianza y ver los 5 mejores. Para ello calculamos la varianza para cada dimensión de los descriptores asociado a cada cluster y hacemos la media de las varianzas (línea 18). Este valor es el asociado a la varianza del cluster en cuestión. Finalmente, sólo debemos ordenar las varianzas (línea 21) y coger los 5 clusteres de menor varianza (línea 22) y visualizar los 20 parches asociados a los 20 descriptores que habíamos obtenido antes de dichos clusters.

Este proceso se ha hecho para los 3 vocabularios obtenidos. Para el vocabulario de tamaño *500* los mejores clusters han sido *148, 9, 178, 498, 167*. Para el vocabulario de tamaño *1000* los mejores clusters han sido *572, 256, 729, 763, 731*. Para el vocabulario de tamaño *5000* los mejores clusters han sido *1578, 689, 1109, 533, 886*. Vamos a ver los parches.

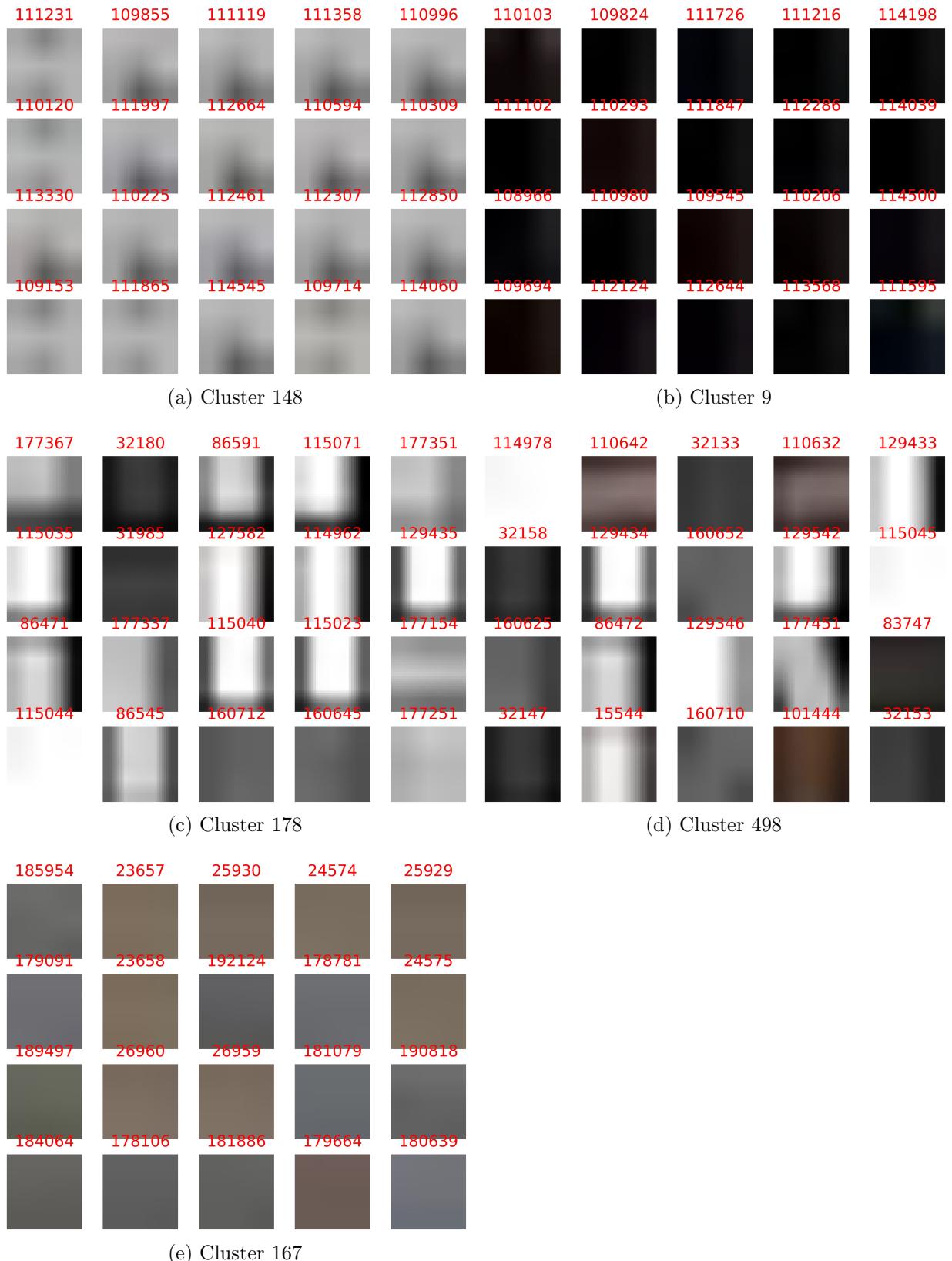


Figura 3.1: Vocabulario de tamaño 500.

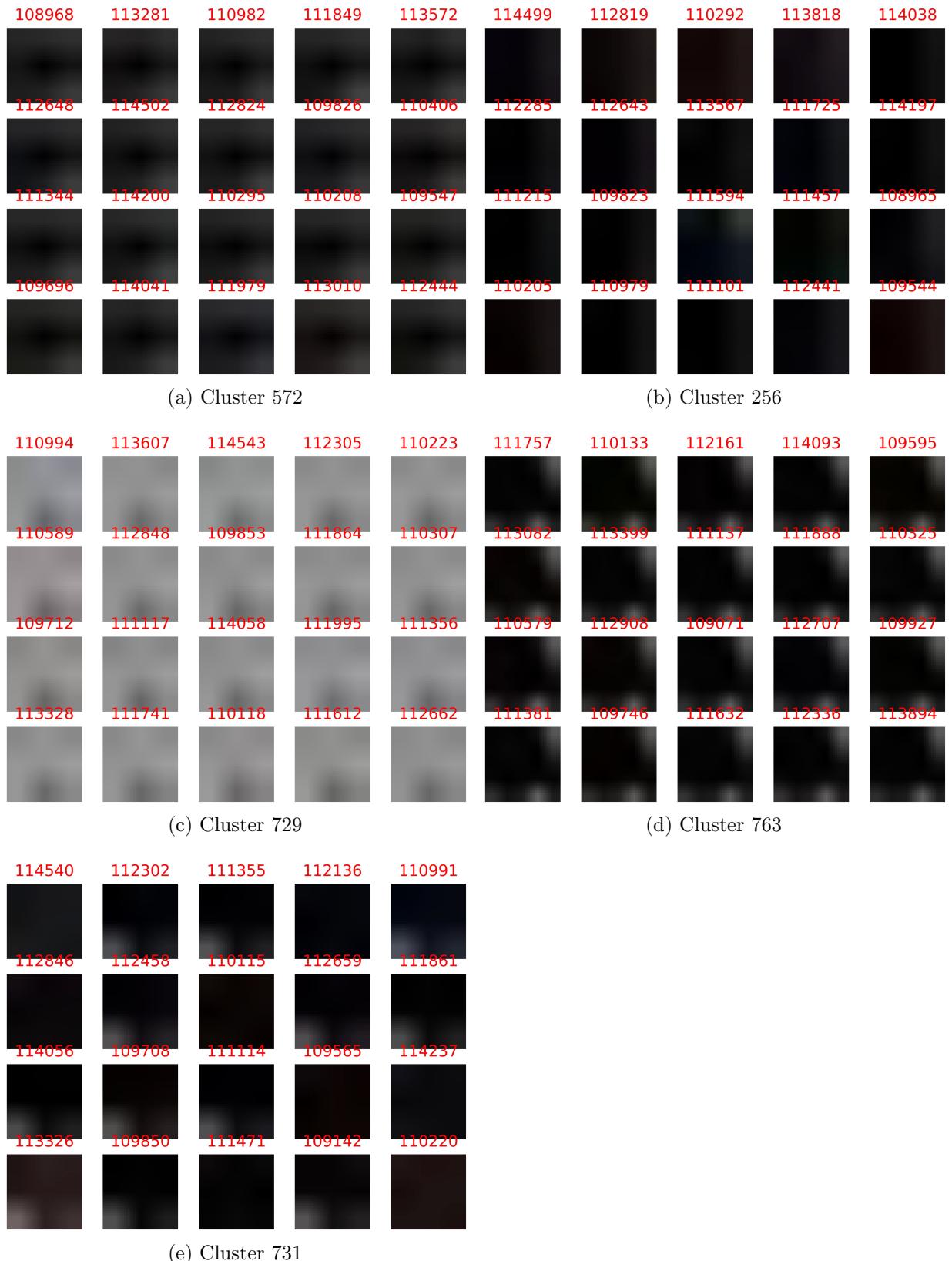


Figura 3.2: Vocabulario de tamaño 1000.

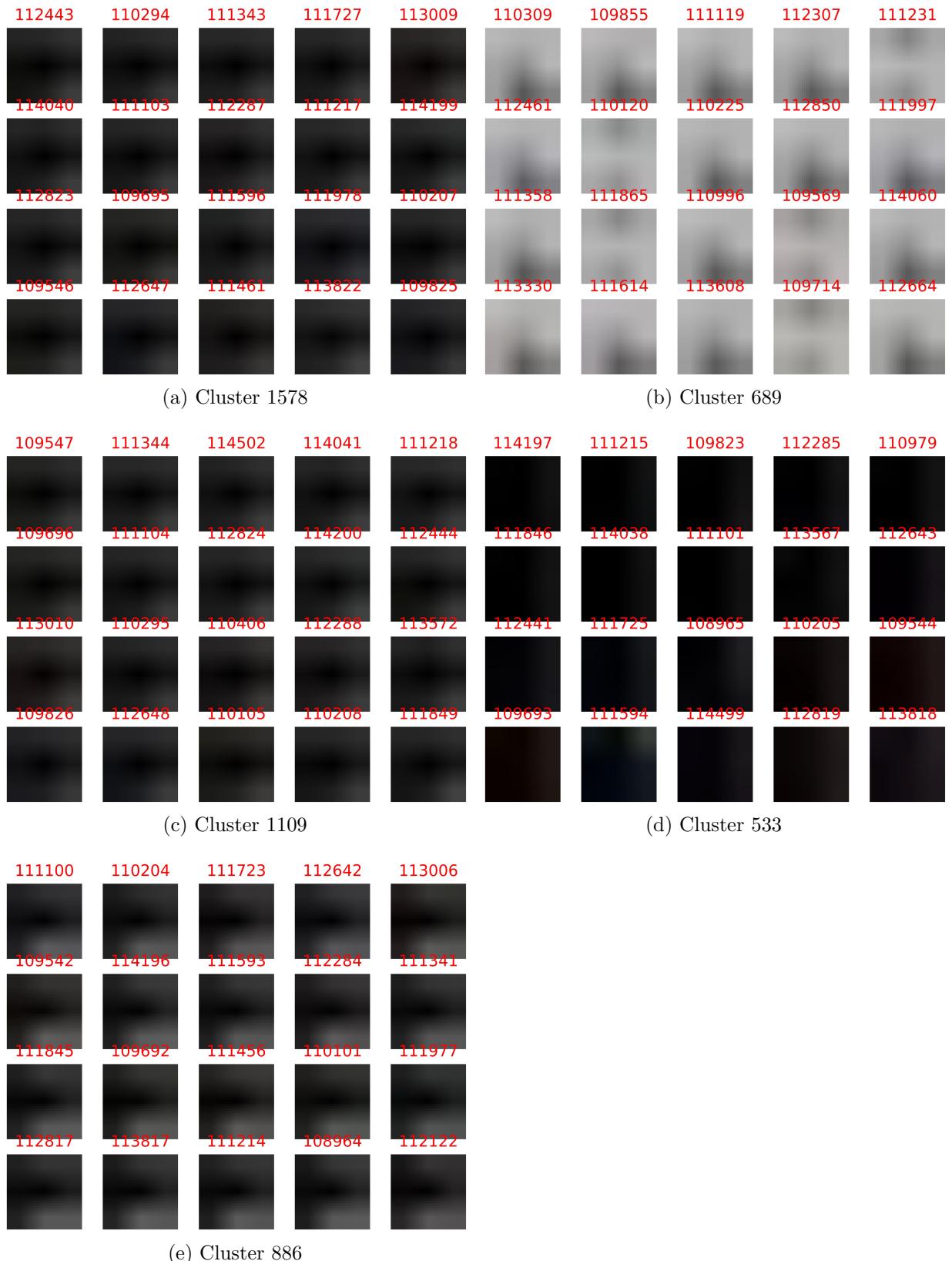


Figura 3.3: Vocabulario de tamaño 5000.

Para cada cluster podemos ver representado sus parches y en rojo el identificador del descriptor correspondiente. Para los vocabularios de tamaño 1000 y 5000 obtenemos que los 20 parches de cada cluster son bastante similares, por no decir idénticos. Sin embargo, para el diccionario de tamaño 500 los parches difieren algo más. ¿A que se debe esto? Cuando usamos un algoritmo de clustering como es el caso, debemos indicarle el número de clusteres que queremos, y esto es algo que no podemos saber a priori. Para el primer vocabulario se ha decidido tener 500 centroides. Viendo los resultados podemos ver que los parches no son del todo similares, esto quiere decir, que el centroide (la palabra) de dicho cluster no es suficientemente representativa de los descriptores de su cluster. Esto quiere decir que nos hemos quedado cortos en el número de cluster empleados, ya que si hubiésemos usado más clusteres podríamos refinar un poco más la asignación de descriptores a los clusteres.

Pero, ¿que pasa en los otros dos casos? Pasa justamente lo contrario. El número de clusteres ha sido tan grande que obtenemos clusteres muy similares (por no decir idéntico). Esto sucede porque usando menos clusteres todas los descriptores similares irían a un único cluster, pero el algoritmo los ha separado en distintos ya que le hemos exigido un número demasiado alto de clusteres a obtener.

Viendo los resultados obtenidos, podemos ver lo difícil que es obtener un diccionario efectivo para la tarea que queremos realizar. También podemos lo artístico que es este proceso y que depende altamente del problema y el conjunto de datos al que nos enfrentemos.

4. Recuperación de imágenes.

La idea de este ejercicio es montar un fichero invertido de palabras que, posteriormente, dada una imagen de consulta nos permita recuperar las imágenes más similares. Este método también está basado en la extracción de puntos SIFT y de descriptores, así que nuestra primera tarea es extraer los descriptores correspondientes:

```

1 def compute_descriptors():
2     descriptor = cv2.xfeatures2d.SIFT_create(nOctaveLayers=4, contrastThreshold
3                                                 =0.000000001)
4
5     descriptors = []
6
7     def atoi(text):
8         return int(text) if text.isdigit() else text
9
10    for dirName, subdirList, fileList in os.walk("./data"):
11        for fname in sorted(fileList, key=lambda x: [atoi(c) for c in re.split('(\d+)', x)]):
12            img = cv2.imread('./data/' + fname)
13            kp, desc = descriptor.detectAndCompute(image=img, mask=None)
14            descriptors.append(desc)
15
16    return descriptors

```

La idea es la misma que hemos desarrollado en la práctica anterior, pero lo hacemos

para cada imagen disponible. Para ello recorremos el directorio `./data` para iterar sobre todas las imágenes (líneas 6-10) y calculamos los descriptores para cada imagen y los guardamos.

Una vez tenemos los descriptores, pasamos a construir el fichero invertido, lo cuál hacemos con la siguiente función:

```

1 def inverted_file(dictionary):
2     descriptors = compute_descriptors()
3     file = collections.defaultdict(list)
4     histograms = []
5
6     for d, img in zip(descriptors, range(len(descriptors))):
7         similarity = np.dot(d, np.matrix.transpose(dictionary))
8         norm = np.apply_along_axis(np.linalg.norm, 1, dictionary)
9         h = Counter(np.argmax(similarity / norm, axis=1))
10        histograms.append(h)
11        [file[k].append(img) for k in h]
12
13    return dict(file), histograms

```

La idea es bien sencilla si entendemos lo que estamos haciendo. Como ya hemos comentado, lo primero que hacemos es obtener todos los descriptores para las imágenes (línea 2). A continuación, para cada conjunto de descriptores, calculamos la similitud de cada descriptor a todas las palabras del diccionario. Si ponemos el diccionario y el conjunto de descriptores de forma matricial, podemos realizar este cómputo de forma eficiente haciendo una simple multiplicación de matrices (línea 7). La fórmula que estamos aplicando para calcular la similitud entre un descriptor, d , y una palabra, w , es la siguiente:

$$sim(d, w) = \frac{\langle d, w \rangle}{\|d\| \|w\|}$$

El producto de las matrices nos calcula el numerador de dicha matriz, luego debemos normalizar. Para ello calculamos las normas de las palabras. Tras realizar la multiplicación de matrices, vamos a tener una matriz con `num_plabras_dicc` columnas y `num_descriptores` filas. La idea es ver a qué palabra debemos asociar cada descriptor. Para ello, diremos que el descriptor d está asociado a la palabra w si la similitud de d y w es la mayor para cualquier otra palabra. Si nos fijamos en la idea, a la hora de seleccionar la palabra para cada descriptor, lo vamos a hacer por filas, por lo tanto, la norma del descriptor no es necesaria, ya que dentro de la misma fila dividiríamos por lo mismo. Este proceso de normalización se hace en la línea 8.

Una vez tenemos la norma, creamos el histograma de las correspondencias entre los descriptores y las palabras. Para ello vamos a usar la clase `Counter` del módulo `collections` (línea 9). En esta línea también realizamos la normalización.

Una vez tenemos el histograma de las palabras, debemos añadir los resultados a nuestro fichero invertido. Este va a tener la estructura de un diccionario, donde la llave es el índice de la palabra y el valor es la lista de imágenes en las que aparece dicha palabra.

Por lo tanto, recorremos el histograma y, para las palabras distintas de cero (la clase *Counter* sólo almacena los valores mayores que cero) y añadir el identificador de la imagen a la lista de imágenes correspondiente (línea 11). Finalmente, devolvemos el fichero invertido y el conjunto de histogramas obtenido a lo largo del proceso.

Una vez tenemos el fichero invertido listo, vamos a realizar el proceso de consulta. Esto lo hacemos con la siguiente función:

```

1 def query(dictionary , file , histograms , img):
2     descriptor = cv2.xfeatures2d.SIFT_create(nOctaveLayers=4, contrastThreshold
3                                                 =0.0000001)
4     kp, desc = descriptor.detectAndCompute(image=img, mask=None)
5
6     similarity = np.dot(desc , np.matrix.transpose(dictionary))
7     norm = np.apply_along_axis(np.linalg.norm, 1, dictionary)
8     h = Counter(np.argmax(similarity / norm, axis=1))
9
10    images = []
11    h_dict = dict(h)
12
13    for k in h_dict:
14        try:
15            images.append(file[k])
16        except KeyError:
17            pass
18
19    flatten_images = [p for row in images for p in row]
20    h_img = Counter(flatten_images)
21
22    h_img_sorted = sorted(dict(h_img).items(), key=lambda x: x[1], reverse=True)
23    n = 100 if len(h_img_sorted) > 100 else len(h_img_sorted)
24    selected_imgs = [e[0] for e in h_img_sorted[0:n]]
25
26    zeros_h = np.zeros(len(dictionary))
27
28    for k in h:
29        zeros_h[k] = h[k]
30
31    similarities = []
32    zeros_query = np.zeros(len(dictionary))
33
34    for img in selected_imgs:
35        for k in histograms[img]:
36            zeros_query[k] = histograms[img][k]
37
38        similarities.append((img, sum(zeros_query * zeros_h) / (np.linalg.norm(
39            zeros_query) * np.linalg.norm(zeros_h))))
40
41    sorted_similarities = sorted(similarities, key=lambda x: x[1], reverse=True)
42    n = len(sorted_similarities) if 5 > len(sorted_similarities) else 5
43    return [e[0] for e in sorted_similarities[0:n]]
```

La idea de este proceso es el mismo que el usado para la construcción del fichero. Extraemos los descriptores, calculamos las similitudes y sacamos el histograma asociado a la imagen de consulta. Este proceso lo podemos ver en las líneas 2 al 7. A continuación, recuperamos la lista de las imágenes que contienen las palabras que contiene nuestra imagen de consulta. Estas listas de imágenes las obtenemos del fichero invertido (líneas 9-16). Una vez tenemos todas las imágenes, construimos un histograma para ver qué imágenes aparecen más (línea 19). A continuación, ordenamos dicho histograma (línea 21) para quedarnos con las 100 imágenes con más ocurrencias (línea 22), ya que las

imágenes que aparecen más son las que tienen más palabras en común con la imagen de consulta y por lo tanto es más probable que sean más similares a la de consulta.

Una vez tenemos las imágenes, pasamos a comparar sus histogramas de palabras con el histograma de la imagen de consulta. Para comparar los histogramas, vamos a convertirlos a vectores (para añadir las palabras con cero ocurrencias) y así poder compararlos correctamente (líneas 25-35). Para comparar los histogramas vamos a usar el coseno del ángulo que forman, de tal forma que si el coseno es 1, las imágenes son la misma. Para calcular el coseno, aplicamos la siguiente fórmula:

$$\cos\alpha = \frac{\langle \text{histograma}_1, \text{histograma}_2 \rangle}{\|\text{histograma}_1\| \|\text{histograma}_2\|}$$

Este cálculo lo hacemos en la línea 37. Una vez tenemos la similitud, simplemente debemos ordenarlas de mayor a menor similitud (línea 39) y quedarnos las cinco mejores (líneas 40 y 41). Veamos los resultados usando los 3 vocabularios:



Figura 4.1: Tamaño 500.

Query: img 13



Image: 13



Image: 5



Image: 8



Image: 133



Image: 144



Figura 4.2: Tamaño 1000.

Query: img 13



Image: 13



Image: 130



Image: 91



Image: 142



Image: 90



Figura 4.3: Tamaño 5000.

Podemos ver que, una vez más, el diccionario usado influye bastante en los resultados obtenidos. Para el tamaño 500 tenemos unos mejores resultados que usando los demás tamaños. Veamos que pasa ahora con otra imagen de consulta:

Query: img 95



Image: 95



Image: 96



Image: 111



Image: 86



Image: 133



Figura 4.4: Tamaño 500.

Query: img 95



Image: 95



Image: 110



Image: 96



Image: 111



Image: 89



Figura 4.5: Tamaño 1000.

Query: img 95



Image: 95



Image: 96



Image: 99



Image: 91



Image: 104



Figura 4.6: Tamaño 5000.

Podemos ver que para la imagen 95 si obtenemos buenos resultados independientemente del vocabulario usado. Veamos ahora un caso en el que los resultados son siempre malos:

Query: img 128



Image: 128



Image: 130



Image: 147



Image: 157



Image: 56



Figura 4.7: Tamaño 500.

Query: img 128



Image: 128



Image: 130



Image: 129



Image: 35



Image: 33



Figura 4.8: Tamaño 1000.

Query: img 128



Image: 128



Image: 130



Image: 90



Image: 96



Image: 33



Figura 4.9: Tamaño 5000.

En este caso, como ya he comentado anteriormente, los resultados son malos independientemente del vocabulario usado.

Pero, ¿a que se debe esto? El problema está en la base de todo el proceso, que son los descriptores. Los descriptores se basan en los gradientes, si los gradientes no son lo suficientemente buenos, los resultados no son del todo apropiados. Recordemos que los gradientes buscan cambios bruscos en las derivadas y, por lo tanto, si las imágenes de consulta no tienen suficientes cambios bruscos, será más difícil obtener unos buenos gradientes para realizar el proceso. Pero si tenemos grandes cambios como sucede por ejemplo en la imagen 95 (debido a la camisa azul y blanca del personaje), sería más fácil obtener unos gradientes y unos resultados buenos.