

PRÁCTICA 2: sEGMENTACIÓN Y RECONOCIMIENTO DE CARACTERES

Visión artificial. Grado en Ingeniería Informática. Universidad Rey Juan Carlos



30 de abril de 2018

Jorge Aranda García

Jose Vicente Bañuls García

ÍNDICE

[1.INTRODUCCIÓN 2](#_Toc512774636)

[2. LOCALIZACIÓN DE LOS DÍGITOS DE LA MATRÍCULA 2](#_Toc512774637)

[3. RECONOCIMIENTO DE LOS CARACTERES LOCALIZADOS 3](#_Toc512774638)

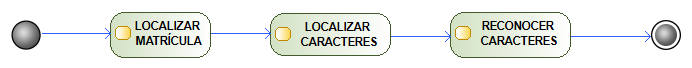
[3.1 Estadísticas 5](#_Toc512774639)

[4. Postproceso 6](#_Toc512774640)

[5. Conclusiones 6](#_Toc512774641)

# **1.INTRODUCCIÓN**

El objetivo de esta práctica es aplicar algunos de los conceptos aprendidos en clase de forma más teórica, en un caso que bien podría ser un caso real. El problema se resume en lo siguiente: Hay que ser capaz de reconocer matrículas de coches. En éstas, hay que encontrar aquello que pueda parecer un carácter y finalmente, una vez que se ha encontrado, reconocer qué carácter es en concreto.



# **2. LOCALIZACIÓN DE LOS DÍGITOS DE LA MATRÍCULA**

En este apartado se va a explicar el proceso que hemos seguido para encontrar aquellos que parece un caracter dentro de una matrícula. Pero, previo a este paso, hay que localizar la matrícula. Para ello hemos utilizado el clasificador proporcionado en la anterior práctica (matriculas.xml) que además de proporcionar unos resultados bastante aceptables, nos va a reducir el espacio de búsqueda de caracteres, limitando la búsqueda sólo al rectángulo reconocido como matrícula, y a su vez el ruido que se podría generar en un espacio mayor. Para ello se utiliza la función *detectMultiScale* que devuelve las coordenadas de los rectángulos que reconoce como matrícula. Una vez se tienen estas coordenadas, en la imagen original se recorta esta parte para centrar la búsqueda.

C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\mat3.png C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\mat2.png

C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\mat1.png C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\mat4.png

A partir de este momento ya empieza el proceso para encontrar los caracteres como tal. El primer paso que realizamos es una ecualización del histograma. A continuación, a la imagen resultante de esta ecualización se le aplica un filtro blur para reducir el ruido, *cv2.bilateralFilter*. A continuación, y esta es una decisión tomada tras realizar cierta experimentación, la imagen es sometida a ciertos umbralizados que nos ayudarán a obtener los mejores resultados. Cabe destacar que estos umbralizados no son realizados de forma secuencial a la imagen (uno detrás de otro), si no que se aplican de forma independiente y se guardan los resultados. El primero de ellos es un umbralizado común (*cv2.threshold(blur, 0, 255, cv2.THRESH\_BINARY\_INV + cv2.THRESH\_OTSU))* que devuelve un resultado aceptable. El segundo de ellos sirve para aquellos casos en los que los dígitos se encuentran algo más juntos, incluso unidos debido a la resolución de la imagen.

(*th100\_165 = cv2.threshold(blur, 100, 165, cv2.THRESH\_BINARY\_INV*

*foo, th\_separador = cv2.threshold(th100\_165, 0, 255, cv2.THRESH\_OTSU)*)

El último de ellos es el que hemos utilizado para casos más “extremos” puesto que nos da respuestas bastante aceptables en aquellas imágenes en las que los anteriores umbralizados han fallado o no son capaces de reconocer demasiado.

(*foo, th50\_110 = cv2.threshold(blur, 50, 110, cv2.THRESH\_BINARY\_INV)  
foo, th\_extremo = cv2.threshold(th50\_110, 0, 255, cv2.THRESH\_OTSU)*)

Todos estos procesos de umbralizado se realizan con el objetivo de ayudar al siguiente paso, que es el de encontrar los objetos conexos. Para ello hemos usado la función *cv2.findContours* y los consiguientes rectángulos que rodean a los contornos devueltos por *findContours*, gracias a la función *cv2.boundingRect* y utilizamos algunas condiciones para descartar rectángulos que no rodeen contornos prometedores. Las condiciones son: La proporción entre el ancho del rectángulo y el alto no debe ser superior de 0.8 ni inferior de 0.55. Con esta única condición algunos se nos escapaban, como por ejemplo los 1. En consecuencia, decidimos introducir otra condición que es la proporción no sea superior a 0.4 ni inferior a 0.2. Cabe destacar también, que los rectángulos se ordenan en función de su X, para tener la matrícula en un orden natural (1234 BCD).

En este punto tenemos aquello que potencialmente será un caracter, pero que aún necesita un procesado más para poder ser clasificados, que será explicado en el siguiente apartado.

Aquí se exponen algunos de los caracteres:

C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\N.PNG C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\5.png  C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\T.PNG

# **3. RECONOCIMIENTO DE LOS CARACTERES LOCALIZADOS**

Este apartado explicará el proceso seguido para la clasificación de los rectángulos que llegan desde el primer apartado.

El primer punto a desarrollar es el entrenamiento para que el clasificador tenga una base. Se nos ha proporcionado un conjunto de entrenamiento que contiene 250 variaciones de cada uno de los caracteres posibles en una matrícula. A partir de estas imágenes hay que realizar un proceso similar al realizado con las imágenes de test del apartado anterior. Primero se aplica un filtro blur para quitar ruido y después se realiza el umbralizado. Se buscan los rectángulos que engloban al dígito, y aunque presumiblemente solo va a haber uno por imagen, se añaden unas condiciones para algunos casos concretos que nos ocasionaron algún problema. Estas condiciones son: el área del contorno debe ser mayor a 4, y la proporción entre el ancho y el alto tiene que ser inferior a 1. Además, hemos añadido otra condición para detectar solamente un contorno por imagen. Con las dimensiones del rectángulo se recorta éste de la imagen original y se aplica un nuevo umbralizado.

Hay que construir el vector de características y todos ellos deben tener el mismo tamaño. Nosotros hemos decidido cambiar el tamaño de la imagen a 10x10 pero manteniendo el aspect ratio de ésta. Para ello se prefija el valor de la altura de la imagen a 10 y se multiplica por el mismo valor en la x y se añaden columnas de 0 hasta que el tamaño sea 10. Hemos utilizado la función:

*cv2.resize(th4,* ***None****, fx=10 / cols, fy=10 / cols, interpolation=cv2.INTER\_LINEAR),*

Donde cols es la altura de la imagen , y la función pad para el relleno comentado:

*left=10-resized.shape[1]  
resized = np.pad(resized, [(0, 0), (left, 0)], mode=****'constant'****, constant\_values=255)*

Aquí se exponen alguno de los caracteres ya redimensionados:

(No se muestran en su tamaño real de 10x10 porque apenas eran visibles. Tener en cuenta solo recuadro blanco. Se puede observar el relleno a la izquierda)

C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\1r.png C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\0r.png C:\Users\jorge_000\AppData\Local\Microsoft\Windows\INetCache\Content.Word\2r.png

Por último, hay convertir la matriz de esta imagen, en una matriz con una sola fila y 100 columnas (Matriz 10x10, cada uno de los píxeles). Hay que almacenar esta matriz de una sola fila e ir concatenando el resultado del mismo proceso para cada una de las imágenes de los dígitos que tendrá 100 columnas y tantas filas como dígitos se usen para el aprendizaje. En nuestro caso esta matriz es “found” y usamos la función *append* para ir añadiendo filas. Por otro lado, se necesita una matriz en la que se guarde la correspondencia de a qué clase pertenece cada una de las filas de nuestra matriz found. Por ejemplo, la fila 0 se correspondería con la clase 0, la 1 también sería la clase 0, la fila 255 pertenecería a la clase 1 y así sucesivamente.

Una vez que hemos obtenido la matriz con los vectores de características y la matriz de correspondencia que nos indica a qué clase pertenece cada vector, podemos empezar con el entrenamiento de nuestro clasificador. Para ello utilizamos la implementación LDA del paquete scikit (***from*** *sklearn.discriminant\_analysis* ***import*** *LinearDiscriminantAnalysis*) y en concreto el método *fit* de esta clase para entrenar la matriz de proyección. Este método ajusta el modelo LDA de acuerdo al conjunto de entrenamiento y a los parámetros proporcionados. En este momento, nuestro clasificador debería estar preparado para reconocer caracteres. Pero antes de ponernos a probar con las imágenes de test proporcionadas hicimos una prueba intermedia para comprobar el correcto funcionamiento de lo que hasta el momento teníamos. La prueba consistía, básicamente, en llamar al método *predict* del objeto LDA, que simplemente se encarga de clasificar el parámetro proporcionado, con las mismas imágenes de entrenamiento. A modo de resumen, al clasificador le acabábamos de decir: “Esto es un 0, dime qué es”. A pesar de ser una prueba simple, supuso una primera aproximación y un indicio de que estábamos en el buen camino.

Una vez que sabíamos que funcionaba ya podíamos empezar a probarlo con casos reales. Pero estos casos reales, nuestras imágenes de test, son de distinto tamaño, además de los rectángulos reconocidos por cv2.*findContours* y *cv2.boundingRect* y también diferentes a las imágenes de entrenamiento. Por lo tanto, para las imágenes de test hay que realizar el mismo procesado que para las imágenes de entrenamiento, es decir, tienen que ser redimensionadas de la misma forma que las imágenes de entrenamiento. Es decir, hay que hacer que cada rectángulo tenga un tamaño de 10x10 y con relleno de ceros a la izquierda y haber umbralizado la imagen igual.

## **3.1 Estadísticas**

Se van a mostrar unas estadísticas de la ejecución del programa con las imágenes propocionadas en testing\_ocr

Como se puede observar los resultados son bastante buenos puesto que en la inmensa mayoría de matrículas es capaz de reconocer todos los caracteres o muchos de ellos (todos menos 1 o 2, o cambia uno por otro)

# **4. Postproceso**

Tras ejecutar el clasificador varias veces nos dimos cuenta de que los resultados no eran del todo aceptables. Esto nos llevó a tomar diversas decisiones y pruebas para intentar mejorar, como por ejemplo realizar los diferentes umbralizados que se han comentado en las imágenes de test. A partir de los resultados obtenidos con cada uno de ellos, nos quedamos con aquella salida más larga de los tres, puesto que suponemos que con dicho umbralizado ha sido capaz de dividir mejor la matrícula.

Por último, se presenta la salida en un fichero de texto donde se muestran la imagen analizada, las coordenadas del centro de la matrícula, el resultado de la clasificación de los caracteres de la matrícula y la mitad del largo de la matrícula.

Aquí se muestran algunos resultados obtenidos:

Imagen: ./testing\_ocr\frontal\_1.jpg, Centro: (332.0, 106.0), Matricula: 11, Largo/2: 64.0

Imagen: ./testing\_ocr\frontal\_12.jpg, Centro: (222.5, 167.0), Matricula: 9597CNW, Largo/2: 84.5

Imagen: ./testing\_ocr\frontal\_8.jpg, Centro: (226.5, 157.0), Matricula: 1877DTY, Largo/2: 72.5

# 

# **5. Conclusiones**

Tras la realización de esta práctica nos hemos dado cuenta de varias cosas. Lo primero, la importancia que tiene tener un buen conjunto de entrenamiento, tanto en cantidad, para tener muchas versiones diferentes de lo mismo, como en calidad, es decir, que realmene sea útil. Para continuar, la importancia de que la imagen de test sea de buena calidad. Al menos, en nuestro caso, no suele fallar porque confunda un carácter con otro o porque lo cambie sin ningún sentido, aunque pase alguna vez. Normalmente el problema reside en que debido a la calidad de la imagen la función *findContours* no es capaz de reconocer los rectángulos que contienen el carácter. Por último, nos hemos dado cuenta de las diversas maneras que hay de afrontar problemas de este tipo y las dificultades que conllevan, además de todos los posibles parches que se pueden ir añadiendo para mejorar los resultados.