



**Universidad
Rey Juan Carlos**

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Curso Académico 2023/2024

Práctica 1

DETECCIÓN DE SEÑALES VIALES

Cristian Fernando Calva Troya

Luis Ovejero Martín

Jaime Rueda Carpintero

Índice de figuras

1.	Imagen con niebla saturada	6
2.	Corrección de angulo de la imagen	6
3.	Imagen tras aplicar filtro azul	7
4.	Imagen tras aplicar sobel	7
5.	Imagen tras detectar lineas y ángulos de desfase	8
6.	Corrección de angulo de imagen	8
7.	Imagen tras aplicar MSER y dibujar las regiones detectadas	9
8.	Imagen tras dibujar un rectángulo en cada región detectada	10
9.	Imagen tras filtrar los rectángulos del paso anterior.	11
10.	Imagen tras ampliar los rectángulos del paso anterior.	13
11.	Subpaneles detectados en la imagen 6.	13
12.	Panel ideal.	14
13.	Panel ideal redimensionado y filtrado por color.	15
14.	Panel detectado redimensionado y filtrado por color.	15
15.	Cálculo del IoU.	17
16.	Algoritmo NMS.	17
17.	Algoritmo NMS aplicado sobre una imagen.	18
18.	Cartel detectado con valor de confianza.	19

Índice

1.	Introducción	4
2.	Objetivos y metodología	4
2.1.	Descripción del problema y objetivos	4
2.2.	Tecnologías	4
3.	Creación de la aplicación	5
3.1.	Normalización	5
3.1.1.	Corrección de color	5
3.1.2.	Corrección de ángulo	6
3.2.	Detección de regiones de alto contraste	8
3.2.1.	Aplicar el filtro de azul	8
3.2.2.	MSER	8
3.2.3.	Extraer los píxeles de la región en un rectángulo	9
3.2.4.	Filtrar rectángulos por relación de aspecto	10
3.2.5.	Expandir regiones filtradas	12
3.2.6.	Detección de subpaneles dentro de la región del cartel	12
3.2.7.	Coordenadas de los subpaneles en la imagen original	13
3.3.	Correlación por máscaras	14
3.3.1.	Mascara ideal	14
3.3.2.	Filtrado por correlación ideal/detectado	15
3.4.	Eliminación de regiones repetidas	16
3.5.	Validación	18
4.	Conclusiones y trabajos futuros	19
5.	Referencias	19

1. Introducción

En esta práctica, abordamos el desafío de la detección de señales viales utilizando técnicas de procesamiento de imágenes. Nuestro objetivo es desarrollar un sistema capaz de identificar y localizar paneles de tráfico en imágenes capturadas por vehículos en carretera.

Para lograrlo, aplicamos una variedad de técnicas de procesamiento de imágenes, desde la normalización y filtrado de color, hasta la detección de regiones de interés y la correlación por máscaras. Estas técnicas nos permiten preprocesar las imágenes de manera adecuada para extraer las características relevantes y luego identificar los paneles de señalización de tráfico en base a un panel ideal.

2. Objetivos y metodología

2.1. Descripción del problema y objetivos

Se desea construir un sistema para la detección automática de paneles informativos de tráfico de color azul, en imágenes realistas que han sido tomadas desde un coche. Los paneles se han diseñado para que sean fácilmente distinguibles del entorno en cualquier condición de luz, color de fondo y climatología.

El objetivo de esta práctica es crear una aplicación que recibiendo una lista de imágenes sea capaz de determinar si hay o no paneles informativos de tráfico y en caso de haberlos determinar en qué región de la imagen se encuentran.

2.2. Tecnologías

Para el desarrollo de esta práctica hemos usado el lenguaje *Python* y, principalmente la librería de *OpenCV* (*cv2*), entre otras como, *matplotlib*, *numpy*, *os* y *math*.

El código de la práctica ha sido desarrollado en *jupyter notebook*.

3. Creación de la aplicación

3.1. Normalización

Una misma imagen tomada en el mismo lugar variará en función de diversos factores como puede ser la luz, climatología, orientación de la cámara y otros muchos factores. En nuestro caso, el mayor problema con el que nos encontramos es con la niebla y los paneles rotados. Al haber nieble en la escena es posible que nos encontremos con una imagen mucha menos nítida y menos saturada. En cuanto a la orientación al tener un cuadrado recto de la detección que no podemos rotar, es posible que tengamos problemas para posicionarlo. A continuación se explicarán los métodos usados para evitar este ruido adverso.

3.1.1. Corrección de color

Uno de los problemas con los que se puede enfrentar el detector es la iluminación de la escena, en concreto con la iluminación producida por la niebla. Cuando hay niebla se pierde nitidez y saturación en la escena. Objetos que se encuentran un tanto alejados desde la posición desde donde tomamos las fotos van a visualizarse de forma borrosa y sin mucha claridad. Este es un problema para el detector ya que para las imágenes donde aparecía niebla el detector no era capaz de detectar ningún panel. Para ello, aplicamos la siguiente fórmula sencilla para obtener la saturación para un imagen:

$$S = \frac{\sum_{i=0}^p s(i)}{p} \quad (1)$$

Donde:

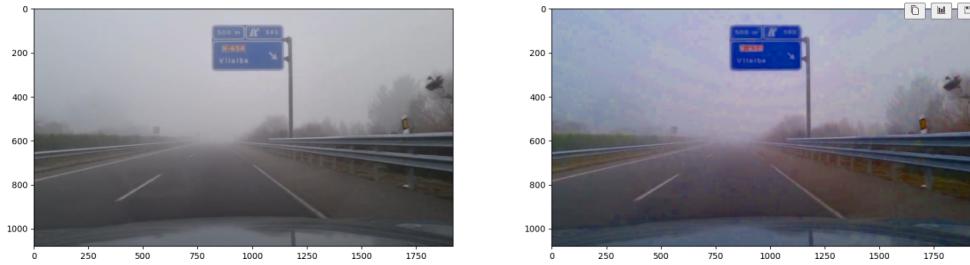
- S = índice de saturación de la imagen
- p = número de píxeles totales de la imagen
- s = saturación del píxel i

Para obtener la saturación de cada píxel, la imagen es transformada al formato HSV. Este formato tiene tres componentes distintas: matiz, saturación y valor. En nuestro caso tendremos en cuenta la saturación ya que es la componente que más diferencia hay entre una imagen normal y una con niebla.

Después de obtener esta media de saturación para cada imagen, establecemos un umbral, donde si una imagen no supera ese umbral entonces tendremos que aplicarle un filtro de aumento de

saturación. Después de ejecutar este algoritmo observamos que la mayoría de imágenes con niebla no eran capaces de superar el umbral de 27, por lo que ese umbral es el establecido. Las demás imágenes sin niebla no pasarán por este filtro. El filtro en cuestión consiste en tomar la saturación de los píxeles y multiplicarlos por 3, para obtener así una imagen mucho más saturada. Así quedaría el proceso de normalización del color, pasando una imagen con niebla a una mucho más saturada, como podemos ver en la **Fig. 1**.

Figura 1: Imagen con niebla saturada



3.1.2. Corrección de ángulo

Figura 2: Corrección de angulo de la imagen

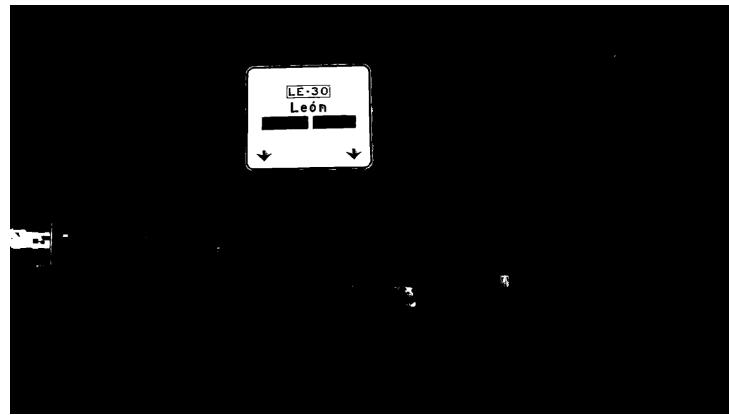


Otro de los problemas presentes a la hora de procesar las imágenes, para su detección, es la posición de los objetos a detectar. En ocasiones, ya sea por el ángulo de la cámara o la misma posición del cartel respecto a nosotros, puede ocurrir un cierto desfase en el angulo deseado. En la **Fig. 2** podemos observar un pequeño ejemplo del problema inicial y el resultado deseado, que nos dará una mejor detección de las secciones. Para poder abarcar el problema hemos optado por realizar una corrección a la imagen completa.

Los pasos a seguir para conseguir el angulo de desfase son los siguientes:

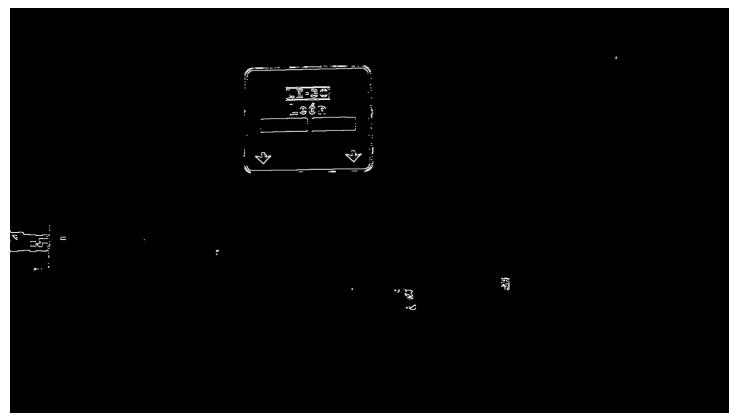
1. Aplicamos una detección de color azul ya que respecto a las diferentes pruebas, ha dado mejores resultados para eliminar las zonas no deseadas y poder centrarnos en las secciones más azules (los paneles deseados.) En la **Fig. 3** podemos ver un ejemplo de los posibles resultados.

Figura 3: Imagen tras aplicar filtro azul



2. Realizamos una detección de bordes horizontales. Ya que haremos la corrección en función de como estén estos orientados. En la **Fig. 4** se ve que conseguimos las secciones deseadas.

Figura 4: Imagen tras aplicar sobel



3. Finalmente con el algoritmo de Hough extraemos líneas presentes en la imagen, dando como resultado las rectas horizontales. En la **Fig. 5** se visualizan las rectas detectadas, y se calcula el angulo de cada uno de ellas haciendo uso de las coordenadas (x, y) y $(x2, y2)$, de estas nos quedaremos con un desfase intermedio. Finalmente con el ángulo de cada imagen, corregimos el giro sobre la imagen original, tal y como se muestra en la **Fig. 6**.

Figura 5: Imagen tras detectar líneas y ángulos de desfase

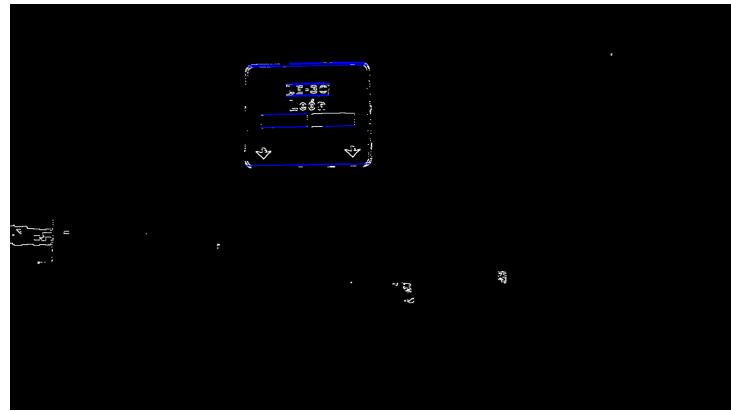
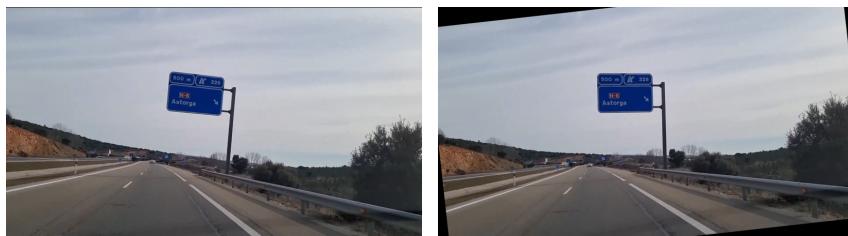


Figura 6: Corrección de angulo de imagen



3.2. Detección de regiones de alto contraste

3.2.1. Aplicar el filtro de azul

Antes de empezar a detectar regiones, vamos a aplicar el filtro de azul sobre las imágenes normalizadas. Esto lo hacemos con la idea de mejorar la detección de los carteles, ya que al aplicar el filtro de azul podemos determinar de forma más eficaz la zona de imagen que puede pertenecer a un cartel.

3.2.2. MSER

Una vez hemos aplicado este filtro a las imágenes ya podemos empezar a detectar regiones sobre las máscaras que hemos generado con el paso anterior.

Para detectar las regiones de alto contraste definimos la función *mser*. Esta recibirá la lista de imágenes normalizadas y la lista de máscaras que hemos creado al aplicar el filtro de azul. Crearemos dos listas que nos servirán para guardar aquellas imágenes sobre las que vamos a dibujar las regiones que detectemos y la otra para guardar las regiones detectadas.

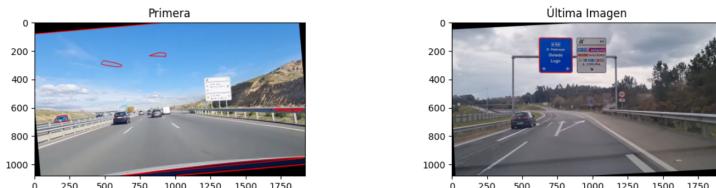
La función va a iterar sobre la lista de imágenes y creará una copia de la imagen normalizada

para dibujas las regiones y una copia de la máscara sobre la que aplicará *MSER*. Antes de aplicar el algoritmo dilatamos la mascara y restamos la máscara original a la máscara dilatada para obtener una imagen de bordes. Tras esto creamos un *MSER* y ajustamos los parámetros para filtrar un poco las regiones que pueda detectar, por ejemplo, haciendo que el área mínima de las regiones que detecte sea 900 y que la máxima sea 5000.

Las regiones detectadas en una imagen se guardaran en la variable llamada *polygons*. Sobre esa lista iteraremos a continuación y para cada elemento de la lista aplicaremos la función *convexHull* de *openCV*, con la finalidad de hallar el casco convexo de un conjunto de puntos. Esto quiere decir que la usaremos para determinar el perímetro de la región que hemos detectado. Guardaremos estos perímetros en la lista *hulls*, que usaremos más adelante para poder dibujar las regiones en la imagen. También guardamos la lista de regiones detectadas en *detected_regions* y finalmente dibujaremos las regiones sobre la copia de la imagen y la guardaremos en la lista que hemos creado al principio para ello. Por último mostramos la primera y última imagen con las regiones dibujadas y devolvemos las listas de las imágenes con las regiones dibujadas y la lista de las regiones detectadas.

Tras ejecutar la función deberían aparecer las imágenes que podemos ver en la **Fig. 7**

Figura 7: Imagen tras aplicar MSER y dibujar las regiones detectadas



En la siguiente celda del *notebook* vemos un bloque de código con un bucle *for* que está comentado, lo podemos descomentar y ejecutar para ver el resultado de nuestra función sobre el resto de imágenes de test. Para pasar las fotos simplemente vamos pulsando cualquier tecla. En caso de querer parar tenemos que pulsar el cuadrado de arriba a la izquierda y cerrar la ventana para terminar la ejecución.

3.2.3. Extraer los píxeles de la región en un rectángulo

Para este paso vamos a definir la función *rectangle_of_regions* que tiene como objetivo dibujar, usando *cv2.boundingRect*, un rectángulo alrededor de las regiones detectadas en el paso anterior. La función recibirá la lista de imágenes normalizadas y la lista de las regiones detectadas

en el paso anterior. Se creará una lista en la que se añadirán las imágenes con los rectángulos dibujados. Iteraremos sobre las imágenes de test y crearemos una copia de cada imagen. Ahora iteraremos sobre las regiones de la imagen sobre la que estamos trabajando y usaremos `cv2.boundingRect`, para obtener el punto superior izquierdo de la región y el ancho y alto de la misma y con ello dibujaremos el rectángulo con `cv2.rectangle`.

Tras dibujar las regiones añadimos la imagen a la lista, imprimimos la primera y la última de estas imágenes y devolvemos la lista donde las hemos guardado.

Al igual que en la región anterior, tenemos una celda de código comentado que podremos usar para poder ver el resultado en el resto de imágenes.

Un ejemplo de los resultados que podemos ver en estas imágenes resultado lo vemos en la **Fig. 8**

Figura 8: Imagen tras dibujar un rectángulo en cada región detectada



3.2.4. Filtrar rectángulos por relación de aspecto

En este apartado se pide que filtremos los rectángulos de las regiones, que hemos dibujado en el paso anterior, por su relación de aspecto con la finalidad de eliminar aquellos que disten mucho de la relación de aspecto de los carteles que se pretenden detectar.

Además de esto hemos hecho que la función encargada de realizar esto también elimine los rectángulos contenidos dentro de otros. Esto lo hacemos porque queremos solo detectar la región del panel azul, no los subpaneles. La idea es encontrar los subpaneles más adelante, pero solo sobre la imagen del cartel en lugar de hacerlo en la imagen completa, así podemos eliminar detecciones basura que pueda llegar a hacer en la imagen completa, pero de esto ya hablaremos próximamente.

Crearemos dos listas, una para las imágenes con las regiones filtradas dibujadas y otra para las regiones filtradas.

Luego iteraremos sobre las imágenes, para cada una de estas vamos a crear una lista para guardar las regiones filtradas de esta imagen y copiaremos la imagen para poder dibujar las regiones en la copia así como hemos anteriormente.

Ahora iteraremos sobre la lista de regiones que pertenecen a esta imagen. Para cada región que hay en la lista vamos a usar `cv2.boundingRect` para obtener el ancho y el alto, con esto podremos calcular la relación de aspecto de las regiones. Nos vamos a quedar con las regiones que tengan una relación de aspecto (ancho/alto) entre 0.6 y 6.5.

Con las regiones que cumplan esto vamos a hacer la comprobación de si están dentro de otra, en caso de que se cumpla y esté dentro de otra, no nos vamos a quedar con ella, solo queremos quedarnos con las regiones más exteriores con la idea de quedarnos con la región del cartel completo, ya que como hemos dicho antes los subpaneles los buscaremos luego.

Iremos agregando las regiones que cumplen ambas condiciones a la lista `filtered_regions` y dibujando las mismas en sus imágenes correspondientes, estas las añadiremos también a `images_filtered_rectangles`.

Finalmente mostraremos la primera ya la última imagen ya que en estas vemos como se eliminan algunas regiones que no cumplen con las exigencias que queremos. Por último vamos a devolver la lista de imágenes y de regiones filtradas.

Al igual que en los pasos anteriores tenemos una celda comentada para poder ver el resultado en el resto de imágenes.

Podemos ver el resultado sobre la **Fig. 8** anterior y la **Fig. 9** que tenemos a continuación.

Figura 9: Imagen tras filtrar los rectángulos del paso anterior.



3.2.5. Expandir regiones filtradas

Al igual que se explica en el enunciado, es posible que haya regiones, sobre las que hemos dibujado un rectángulo, en las que nos haya quedado parte del cartel fuera del rectángulo y para arreglar este pequeño problema, que puede empeorar los resultados en pasos posteriores, tenemos que aumentar el tamaño del rectángulo.

Para solucionar esto tenemos la función *expand_regions*. Al igual que las funciones anteriores recibirá dos listas. Una de las imágenes normalizadas y la otra de las regiones filtradas en el paso anterior.

La función irá iterando sobre las imágenes de la lista que recibe como primer argumento. Para cada imagen iterará sobre sus regiones filtradas y hallará el punto superior izquierdo (valores *x* e *y*), el ancho y el alto de la región. Una vez tiene estos valores en el caso de los valores de *x* e *y* restaremos un valor para que se muevan más hacia arriba a la izquierda. En el caso de la altura y anchura sumaremos un valor porque queremos que sean más grandes.

Dibujaremos el rectángulo de nuevas dimensiones sobre la copia de la imagen normalizada que hemos hecho previamente. Después de eso vamos a crear una pequeña imagen de la región, llamada *pixels_region*. También vamos a guardar en una lista de posibles paneles (*possible_panels_img*) este recorte de la imagen normalizada en la que podría haber un panel. Además guardaremos en forma de tupla las coordenadas de cada región ampliada para usarlas más adelante.

Al igual que hemos hecho en los pasos anteriores vamos a mostrar el resultado de la primera y última imagen y se devolverá la lista de imágenes con las regiones ampliadas dibujadas, otra lista para las coordenadas nuevas de las regiones y la lista de posibles subpaneles.

Al igual que en los apartados anteriores podemos ver el resultado del resto de imágenes al ejecutar la celda de código siguiente.

Un ejemplo sería el que podemos ver en la **Fig. 10**

3.2.6. Detección de subpaneles dentro de la región del cartel

El proceso de filtrado de las regiones detectadas tenía como objetivo llegar a este punto. Según el enunciado de la práctica, esta parte no se pedía. Nuestra idea era trabajar solo con las regiones que podrían ser carteles a la hora de buscar los subpaneles, ya que pensamos que al haber menos elementos en la imagen seríamos capaces de localizar los paneles de manera más efectiva.

Dentro de este apartado vamos a volver a ejecutar acciones parecidas a las que hemos usado

Figura 10: Imagen tras ampliar los rectángulos del paso anterior.



ya, pero en este caso lo haremos sobre los recortes de las imágenes que hemos determinado que pueden ser carteles.

En la función `detect_subpanels` vamos a recibir las imágenes normalizadas y la lista de los recortes de cada imagen. Lo que haremos será, imagen por imagen, volver a usar `mser` para detectar las regiones de los subpaneles. Estas imágenes las vamos a redimensionar con `cv2.resize` para hacerlas más grandes. Les aplicaremos un filtro de azul para binarizarlas y usaremos `cv2.Canny` para hallar la imagen de bordes. La dilataremos para no perder detalles en los bordes y aplicaremos `mser` para detectar los subpaneles. Posteriormente a detectar las regiones vamos a ampliarlas y guardar sus coordenadas.

Finalmente la función va a devolver una lista de recortes de cada región detectada anteriormente, es decir las subregiones dentro de las regiones previas. Estos recortes son posibles subpaneles de las imágenes (`possible_subpanels`). También vamos a devolver las coordenadas de estas subregiones en la lista `subpanel_coords`.

En la **Fig. 11** podemos ver las subregiones detectadas en este paso en la imagen 6.

Figura 11: Subpaneles detectados en la imagen 6.



3.2.7. Coordenadas de los subpaneles en la imagen original

Este es el último paso antes de pasar a la correlación de máscaras. En este paso vamos a corregir las coordenadas que acabamos de obtener de los subpaneles. Esto lo hacemos porque al

aumentar el tamaño de la región sobre la que hemos detectado las subregiones, las coordenadas de la subregión también se ven afectadas por el cambio. Para ello tenemos que corregir las coordenadas en el sentido inverso en el que habíamos aumentado la imagen. En el paso anterior hemos multiplicado por tres el ancho y el alto de la imagen y ahora tendremos que dividir por tres.

Para esto tenemos la función *fix_coords* de manera similar a como lo hemos ido haciendo anteriormente obtendremos las coordenadas de cada posible subpanel detectado en cada una de las imágenes. Una vez tenemos las coordenadas las ajustamos y las volvemos a guardar en una lista que vamos a devolver.

3.3. Correlación por máscaras

3.3.1. Mascara ideal

Figura 12: Panel ideal.

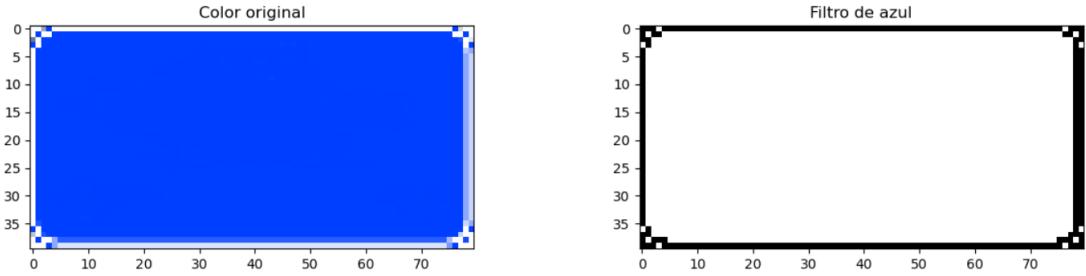


Para la correcta comprobación de que si una imagen ya filtrada por color es un panel deseado, necesitamos un panel de referencia, para poder indicar como debería ser un panel detectado de forma aproximada.

En la **Fig. 12** podemos ver la imagen de la que partimos para realizar esa mascara ideal que utilizaremos posteriormente. Esta imagen inicial se extrae de la Norma 8.1-IC de señalización vertical de carreteras en España [1].

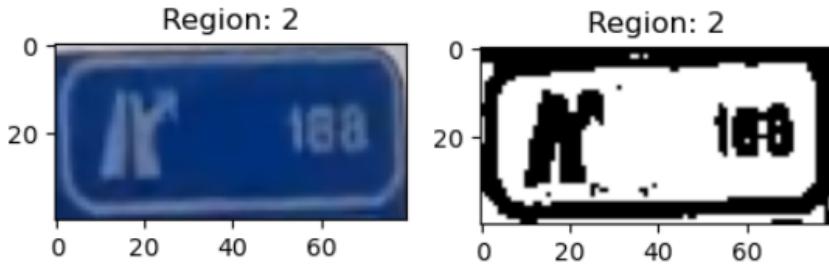
Para poder crear la matriz de 1's (para azules) y 0's (para no azules) hacemos uso de un filtro de color. Antes de poder hacer uso del filtro de color necesitamos cambiar el formato de la imagen de **BGR** a **HSV**. Este cambio es fundamental porque en el espacio de color HSV, el componente Hue permite una separación clara del color azul, independientemente de las variaciones de iluminación y sombra. Esta conversión y filtrado mejoran significativamente el procesamiento y permiten realizar comparaciones directas con un panel de referencia, facilitando la verificación.

Figura 13: Panel ideal redimensionado y filtrado por color.



Ya que nos encontraremos con diferentes paneles de diferentes tamaños, establecemos una tamaño estandar de (80x40) a redimensionar para todos los paneles, tanto detectados como para el ideal. Tras aplicar esta redimensión podemos ya aplicar el filtro de color a nuestro panel, en la **Fig. 13** podemos ver el resultado de este proceso con el panel ideal.

Figura 14: Panel detectado redimensionado y filtrado por color.



Respecto a los paneles detectados en la **Fig. 14** podemos ver un ejemplo de su proceso de redimensionado y filtrado por color. De esta manera ya podríamos realizar una comparación y correlación entre detectado/ideal.

3.3.2. Filtrado por correlación ideal/detectado

Llegados a este punto quedaría decidir que paneles recibidos son lo suficientemente parecidos al panel ideal. Para ellos realizamos:

1. Una suma de cuantos píxeles azules tiene nuestra máscara ideal. Teniendo en cuenta que es una máscara de 0's y 1's queda bajo la siguiente operación:

$$B = \sum_{i=0}^{39} \sum_{j=0}^{79} I(i, j) \quad (2)$$

2. Calcular la cantidad de azules que tiene nuestra detección. De igual forma que

en el paso anterior:

$$B' = \sum_{i=0}^{39} \sum_{j=0}^{79} I'(i, j) \quad (3)$$

3. Calcular un porcentaje de correlación normalizado entre [0,1]. Simplemente haciendo una proporción de B' sobre B , en caso de que B' supere a B descartamos esta detección poniendo su proporción a 0. Quedando:

$$C = \begin{cases} B'/B & \text{si } B' \leq B, \\ 0 & \text{si } B \leq B'. \end{cases} \quad (4)$$

4. Descartamos aquellos valores C que superen un umbral. En este caso se ha puesto uno mínimo de 0'4.

Posterior a estos pasos nos quedarían las secciones más fiables a la detección deseada, pero nos quedaría eliminar posibles detecciones repetidas.

3.4. Eliminación de regiones repetidas

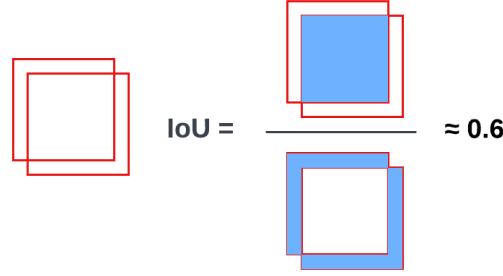
Una vez hemos aplicado todo el proceso del detector y hemos detectado regiones en forma de cuadrados, es posible que tengamos un solapamiento de diversos rectángulos, unos encima de otros. Esto no sería lo óptimo ya que hemos detectado subpaneles, por lo que nunca tendríamos otros subpaneles dentro de estos. Lo mejor sería eliminar los paneles que están dentro de otros y que aporten poco con respecto al rectángulo en el que están contenidos.

El algoritmo escogido es el NMS (Non-Maximum Suppression). Este algoritmo se encarga de eliminar regiones contenidas en otras que tienen una puntuación menor que la que tienen las otras regiones. Se basa en la Intersection over Union (IoU) que se calcula dividiendo la intersección de dos regiones entre la unión entre dos regiones. Podemos ver un ejemplo gráfico en la **Fig. 15**

El valor obtenido se encuentra comprendido entre 0 y 1. Un valor de 1 quiere decir que las dos regiones están perfectamente solapadas y un valor de 0 quiere decir que son dos regiones que no se llegan a tocar nunca.

NMS itera sobre cada uno de los paneles detectados y calcula el IoU sobre esos dos paneles actuales iterados. Si el valor de IoU es mayor que 0 quiere decir que los dos subpaneles están superpuestos. Cuando haya detectado una superposición, entonces se evaluará las puntuaciones de las regiones. Estas puntuaciones las obtenemos en el último paso de la detección a través

Figura 15: Cálculo del IoU.



de la correlación. Escogeremos el panel que tenga mayor puntuación ya que es el panel que mejor se ajusta al panel ideal y eliminamos de la lista la región con menor puntuación. Este proceso se repetirá constantemente con cada uno de los paneles. El pseudocódigo del algoritmo se encuentra representado en la **Fig. 16**

Figura 16: Algoritmo NMS.

Algorithm 1 Non-Max Suppression

```

1: procedure NMS( $B, c$ )
2:    $B_{nms} \leftarrow \emptyset$  Initialize empty set
3:   for  $b_i \in B$  do => Iterate over all the boxes
4:      $discard \leftarrow \text{False}$  Take boolean variable and set it as false. This variable indicates whether b(i) should be kept or discarded
5:     for  $b_j \in B$  do Start another loop to compare with b(i)
6:       if same( $b_i, b_j$ )  $> \lambda_{nms}$  then If both boxes having same IOU
7:         if score( $c, b_j$ )  $>$  score( $c, b_i$ ) then
8:            $discard \leftarrow \text{True}$  Compare the scores. If score of b(i) is less than that of b(j), b(i) should be discarded, so set the flag to True.
9:         if not  $discard$  then
10:           $B_{nms} \leftarrow B_{nms} \cup b_i$  Once b(i) is compared with all other boxes and still the discarded flag is False, then b(i) should be considered. So add it to the final list.
11:    return  $B_{nms}$  Do the same procedure for remaining boxes and return the final list

```

Hemos introducido una pequeña modificación en código para evitar un problema que podíamos tener. El problema en cuestión es que es posible que en algunas ocasiones un panel detectado abarque un poco de región de otro, por lo que el algoritmo lo tomará en cuenta al tener un IoU mayor que 0 y eliminará una región de estas. Esto no es bueno ya que los dos carteles son independientes y no queremos eliminar ninguno de ellos, sino regiones dentro de estos. Para ello, definimos un umbral de IoU de 0.05. Si tenemos un IoU mayor que este umbral significará que tendremos que eliminar una de las regiones ya que es posible que se trate de una región dentro de otra. Así quedaría el algoritmo aplicado en una de las imágenes, como se muestra en

la **Fig. 17**

Figura 17: Algoritmo NMS aplicado sobre una imagen.



3.5. Validación

Para finalizar, hemos agregado en una lista las 20 imágenes para nuestro conjunto de prueba. Son imágenes variadas donde hay imágenes con niebla, con más luz, sin carteles o con carteles pequeños, seleccionadas para comprobar la robustez del detector. Para obtener una mejor retroalimentación de la eficacia del detector, hemos anotado manualmente los paneles que realmente aparecen en la escena para luego compararlos con los paneles detectados. Para la anotación de los carteles hemos empleado la página web [Roboflow](#) ya que permite anotar regiones de forma manual muy rápidamente. La información anotada se devolverá en un formato JSON donde aparecerán las coordenadas del punto de arriba a la izquierda y el ancho y el alto de cada uno de los rectángulos. El archivo donde se encuentra esta información es el llamado *anotaciones.json*. Para evaluar la proximidad del cartel detectado con el real usaremos la métrica IoU. Para cada uno de los carteles detectados calcularemos el valor de IoU para cada uno de los carteles reales y nos quedaremos con el valor máximo de este. De este modo cada cartel tendrá su propio valor anotado. Hay que tener en cuenta que este valor puede ser bajo ya que los carteles reales los anotamos en la imagen original, es decir, sin rotarla. Los carteles detectados se encuentran ya en la imagen rotada y por eso el IoU puede resultar bajo. Para finalizar, representaremos los carteles detectados junto con el valor IoU, en la **Fig. 18**

Figura 18: Cartel detectado con valor de confianza.



4. Conclusiones y trabajos futuros

Como conclusiones después de haber terminado este trabajo podemos decir que lo más complicado ha sido la parte de detectar los carteles con *MSER*, siendo especialmente complicada la tarea, en aquellas imágenes que tenían bastante niebla o que había demasiada luz, haciendo que el color de los carteles fuese más difícil de detectar.

En la mayoría de los casos de test hemos podido comprobar que los resultados son buenos, incluso en aquellas imágenes en las que había niebla, o que estaban torcidas.

Para mejorar estas detecciones se podría hacer usando un modelo de aprendizaje automático, ya que en general suelen dar mejores resultados que este tipo de algoritmos.

5. Referencias

- [1] <https://www.fomento.gob.es/az.bbmf.web/documentacion/pdf/re3723.pdf>
- [2] <https://blog.roboflow.com/how-to-code-non-maximum-suppression-nms-in-plain-numpy/>
- [3] <https://builtin.com/machine-learning/non-maximum-suppression>
- [4] <https://roboflow.com/>
- [5] <https://github.com/adheeshc/Traffic-Sign-Recognition/blob/master/README.md>