PRÁCTICA 3

Miembros del grupo

Miguel Cortecero Torres - 52017329T

Karen Wiznia Fresco - 51114786T

Eduardo de la Torre Díaz - 48146805P

Parte 1

Algoritmo implementado

Para la parte 1 de la práctica se ha hecho uso del paradigma de clases y la estructura resultante consta de:

Práctica1.py: que es la clase principal desde donde serán llamados los métodos de las diferentes clases

KeyPoint: clase que se basa en la estructura de un keyPoint de openCV pero añade un campo más que es distanceToCenter. Este atributo contendrá, el módulo, el vector, el ángulo y el punto del centro de la imagen del vector de votación asociado al keyPoint

Operations: clase que contiene métodos de apoyo, en este caso únicamente posee la función del cálculo al centro.

Training: contiene los métodos para leer las imágenes de entrenamiento y entrenar nuestro sistema con ellas.

Processing: contiene los métodos que nos permite obtener la ubicación de un coche a partir del detector entrenado anteriormente.

Training

Para la etapa de entrenamiento leeremos todas las imágenes que tengamos en la carpeta "training". De cada imagen hallaremos sus puntos de interés usando un detector ORB (*orb = cv2.ORB(nfeatures=kpNum, nlevels=4, scaleFactor=1.3)*). En dicho detector le indicaremos cúantos puntos de interés queremos que nos obtenga de cada imagen, que en nuestro caso serán 150, el número de niveles en los que queremos que detecte y con qué factor de escala. Esto es importante para poder luego distinguir imágenes con distintos tamaños de coches.

Una vez obtenidos los puntos de interés, hallaremos el vector que los une con el centro y los incluiremos en un objeto de tipo KeyPoint. Además, de cada imagen nos guardaremos un array global, en su posición correspondiente, tanto los puntos de interés como los descriptores, para su posterior uso en la etapa de procesamiento/testing.

Además, de cada imagen crearemos un índice de búsqueda (flannBasedMatcher) que contendrá los descriptores asociados a la imagen correspondiente, y ese índice se guardará en un array global que contendrá todos los índices de todas las imágenes.

Processing

Una vez entrenado el detector procederemos a leer las imágenes de prueba para buscar la posición del coche en la imagen.

En esta parte tendremos una máscara de acumulación donde iremos almacenando votos encontrados para la imagen actual.

Para ello, de cada imagen de prueba habrá que sacar los puntos de interés y los descriptores usando un ORB de la misma manera que en entrenamiento. Una vez hallados, crearemos una máscara intermedia en la que se añadirán los votos que provenientes de las imágenes de entrenamiento.

Para llegar hasta ello primero debemos buscar los k-vecinos más cercanos a cada punto de interés encontrado en la imagen de testing usando la función *flann.knnMatch(descProcessingArray,* self.*globalDescArray[index],* k=*5)*. Una vez obtenidos los vecinos tendremos que añadir los votos a la máscara intermedia. Para hacer esto deberemos comparar cada punto de interés de la imagen de prueba con cada punto de interés de las imágenes de entrenamiento, teniendo en cuenta su ángulo y su distancia al centro. Realizando las operaciones pertinentes obtendremos el vector de votación, que nos indicará el punto donde se produce un voto. Es aquí entonces donde tendremos que anotarlo en la máscara intermedia, descartando aquellos votos que se produzcan fuera de la imagen.

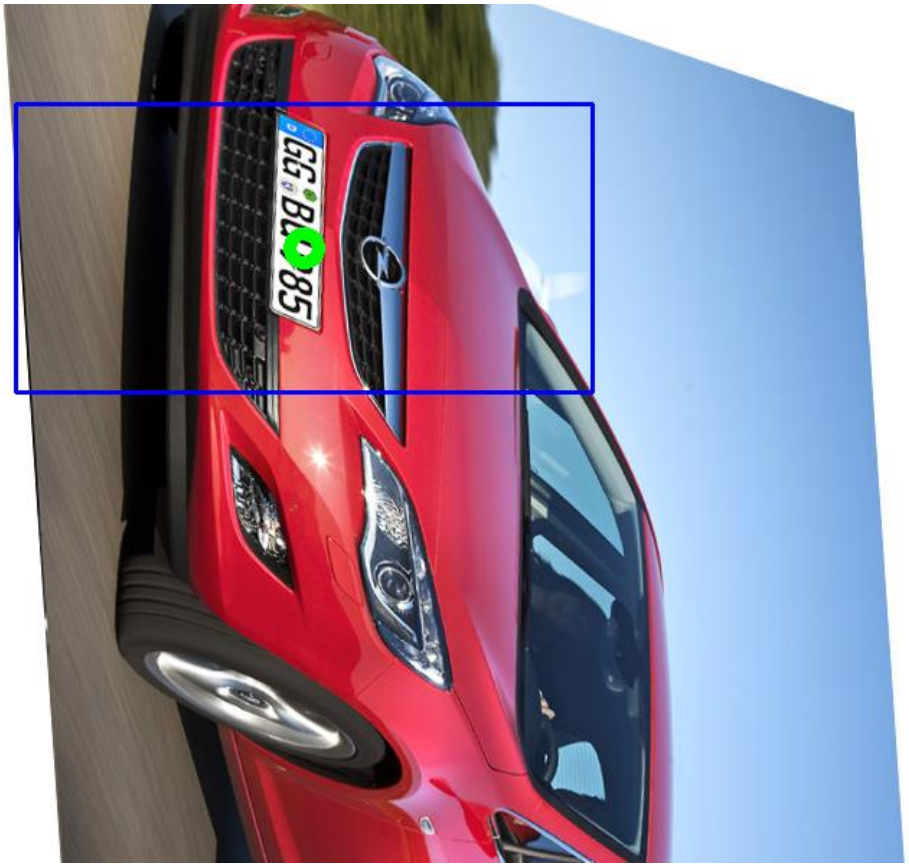
Este proceso se repite comparando cada imagen de prueba con todos los datos obtenidos en entrenamiento, integrando los votos de cada máscara intermedia en la máscara de acumulación final. Por lo tanto, al final del proceso obtendremos una máscara de acumulación que será una matriz de valores, donde el valor máximo de esa matriz representará donde se encuentra el coche. (*cv2.minMaxLoc*)

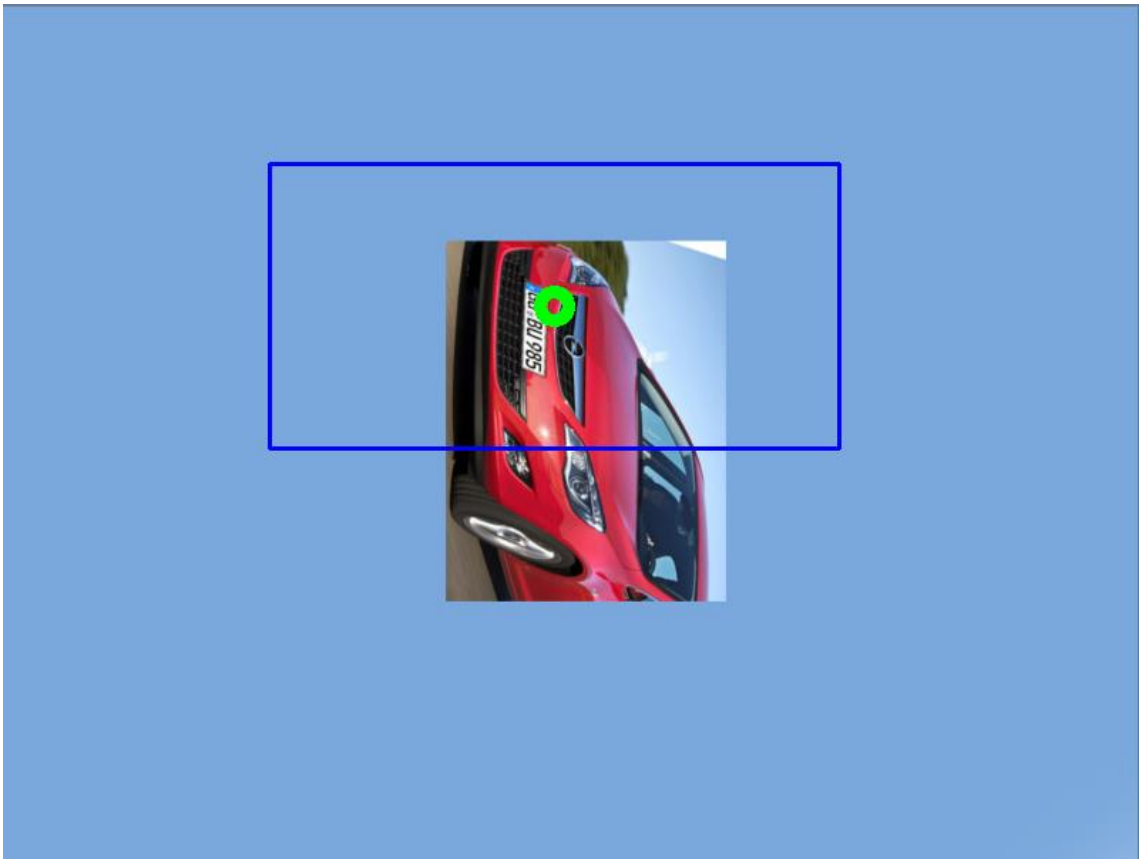
Finalmente dibujamos el punto y el área donde se encuentra el coche con los siguientes métodos: *cv2.circle(processingImage, maxLoc, 10, (0,255,0),* thickness=*7,* lineType=*8,* shift=*0)* *cv2.rectangle(processingImage, pt1, pt2, (255,0,0),* thickness=*2,* lineType=*8,* shift=*0)*

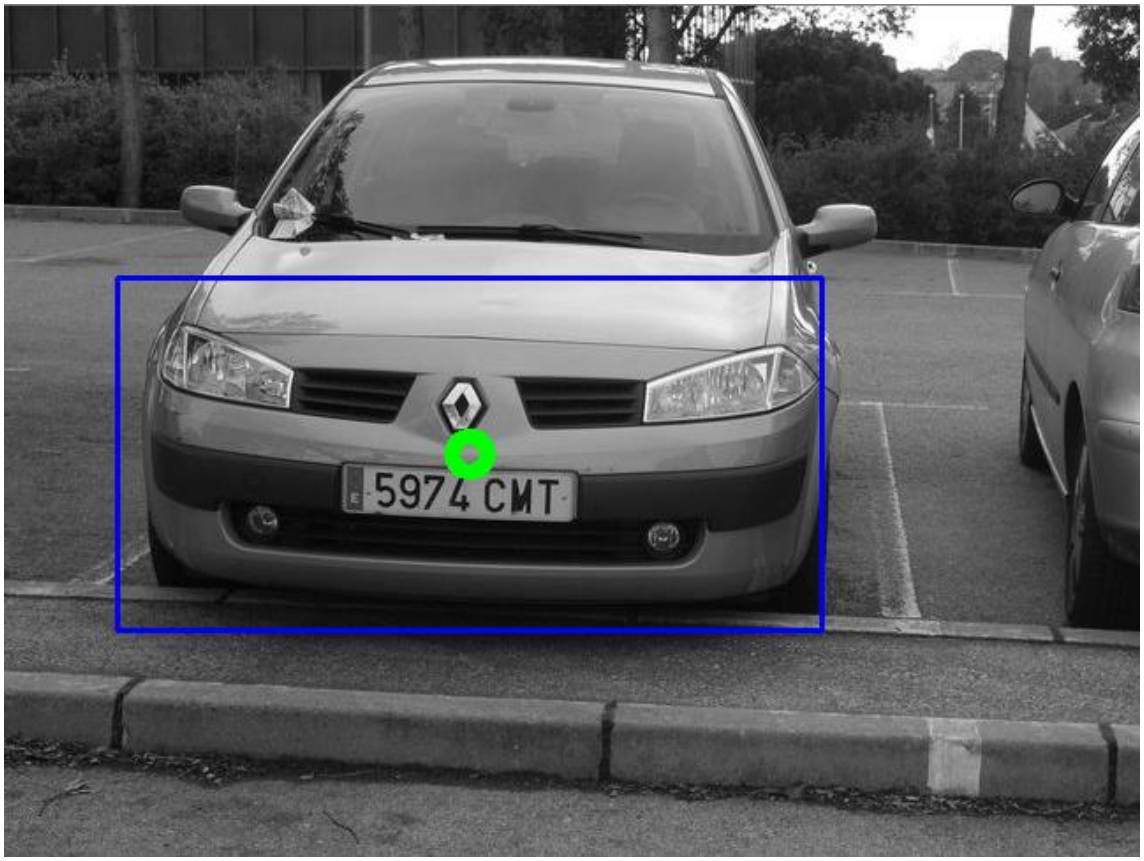
NOTA: en nuestro caso no hemos considerado necesaria hacer la rotación de los vectores de votación encontrados pues estamos usando un detector ORB (que está basado en FAST y BRIEF), que es invariante a los cambios de rotación. http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_orb/py_orb.html

Ejemplos en ejecución

Para la prueba de funcionamiento se han probado imágenes tanto las imágenes suministradas como otras a las que les hemos aplicado alguna transformación (cizallado, rotación, disminución de tamaño).

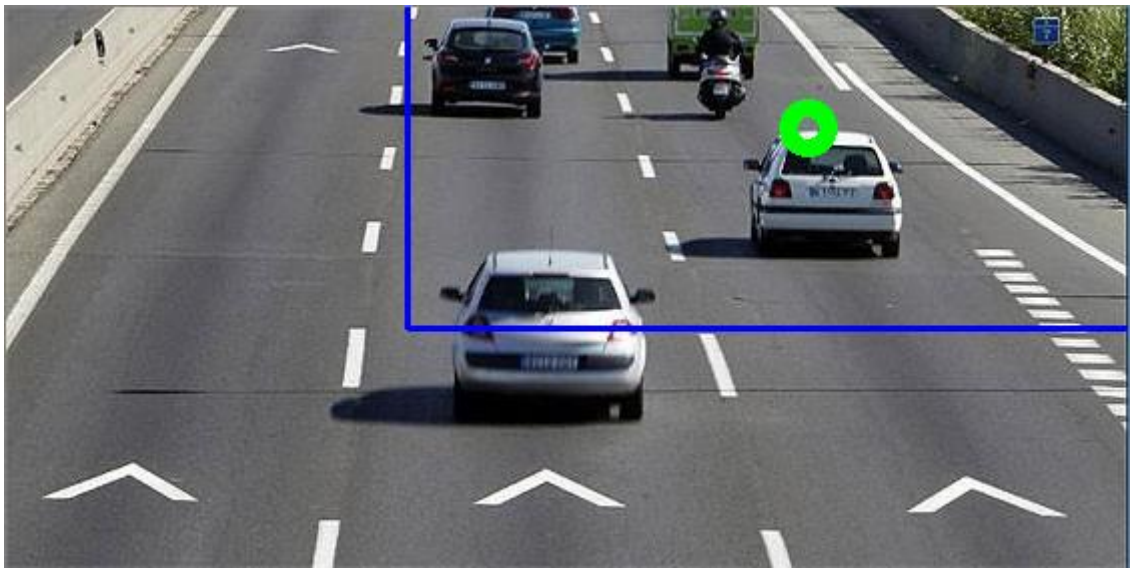





5974 CMT


6079 DXR

En imágenes con varios coches sólo es capaz de mostrar la ubicación de uno de ellos.



Cuando la imagen es borrosa suele fallar al encontrar el coche correctamente.

Estadísticas

Imágenes de entrenamiento: 33

Imágenes de prueba: 33

Número de vecinos buscado: 5

Número de puntos de interés buscado: 150

De 33 imágenes de prueba se han obtenido 27 en las que el coche se ubica al menos un coche correctamente centrado, con un ligero margen de error. Es decir, aproximadamente, un 82% de acierto.

Parte 2

Para la parte 2 de la práctica la estructura contiene un único fichero llamado parte2.py donde se realiza un algoritmo que permite la detección de una posición del coche en una imagen. Para ello la estructura de la clase está organizada de la siguiente manera:

En primer lugar con la utilización del método cv2.CascadeClassifier somos capaces de cargar los diferentes clasificadores suministrados en esta práctica. Por un lado el de coches, cuya información se guardará en una variable denominada *cascadeCars* y por otro lado, el de matrículas que se guardara en una variable denominada *cascadeMatriculas.*

Una vez realizado esto, se procederá a explicar cada uno de los métodos implementados en esta clase:

- Detect: Este método se encarga de buscar los diferentes rectángulos donde cree que está la imagen. Para poder realizar esto, se utilizó el método *cascade.detectMultiScale* con el fin de detectar los diferentes objetos, de diferentes tamaños en la imagen. Para esto se utilizó un factor de escala de 1.2. Finalmente esto, devuelve una lista vacía, en el caso de que no encuentre ningún rectángulo que se ajuste a esos parámetros, o una lista con los rectángulos en caso contrario.
- draw_rects: Se ha implementado como método auxiliar, para ser utilizada en clases posteriores con el fin de permitir pintar los diferentes rectángulos.
- detectarCoches: este método se encarga de la detección de los coches con el detector de Haaris. El propósito de este método es ir leyendo las diferentes imágenes guardadas en una variable llamada *cascadeCars* y mediante la función *detect* (explicada anteriormente) intentar detectar los diferentes rectángulos que conforman la imagen. Una vez detectados cada uno de estos rectángulos, se dibujan los mismos junto con la imagen.
- detectarMatriculas: este método cumple la misma funcionalidad que *detectarCoches* solo que esta vez para las matrículas. De este modo, en el método *detect,* lee las imágenes almacenadas en la variable *cascadeMatriculas.* El funcionamiento del mismo, es exactamente el mismo que el anterior.

Ejemplos en ejecución

Para la realización de las pruebas correspondientes, se han utilizado las imágenes suministradas. A continuación se muestra el resultado de aplicar el algoritmo anteriormente explicado a alguna de estas imágenes.


FIAT
E 1512 DLC


FIAT
E 1512 DLC


1040 CLR


1479 BHW

Estadísticas

Imágenes de entrenamiento: 33

Imágenes de prueba: 33

De 33 imágenes de prueba se ha conseguido detectar el coche en 24 de ellas, lo que supone un 75% de aciertos.

De las 33 imágenes, se ha conseguido detectar la matrícula en 18 de ellas, es decir, ha acertado en un 54.5% de aciertos.

Parte 3

Para la parte 3 se ha usado el mismo método de detección que la parte 3 por lo que gran parte de la implementación es muy parecida.

En esta parte ya no se hace uso de imágenes para la detección del coche y la matrícula sino que se usan videos como elemento de entrada. En nuestro caso se han hecho las pruebas con los dos videos suministrados junto con un vídeo propio. Cada vídeo tiene distintas características por lo que permite comprobar exhaustivamente el funcionamiento del detector.

Como se ha comentado, en este caso se va a aplicar el detector haar a los vídeos. Para ello, de cada vídeo se irán leyendo sus distintos fotogramas y se irán detectando los coches y matrículas en cada fotograma. Esto se consigue mediante la función de openCv VideoCapture:

- Primero crearemos un objeto del tipo VideoCapture de la siguiente manera *cascade = cv2.CascadeClassifier(cascade_file)*.
- Con este objeto iremos leyendo los distintos fotogramas del vídeo, a los que les iremos aplicando los detectores, de la manera en que se explicó para la parte 2.
  Para leer los fotogramas usaremos el método read del objeto de tipo CascadeClassifier, que nos devolverá el frame actual y un valor que nos permite saber cuándo hemos terminado de leer frames. El uso será el siguiente *ret, frame = cap.read()*

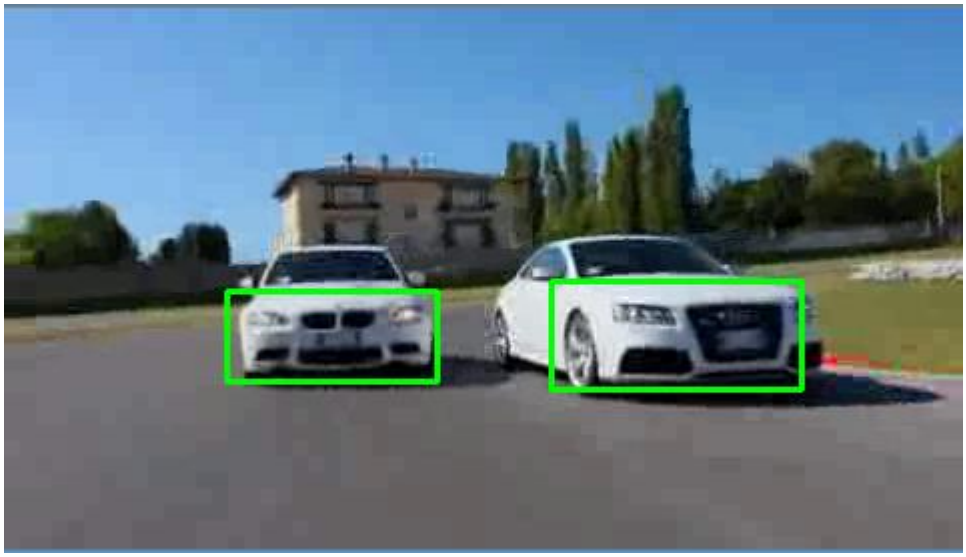
  Además, es posible parar el vídeo en algún punto por si se quieren observar más detalles en algún momento pulsando la tecla “p”.
  También se permite pasar al siguiente vídeo pulsando la tecla “q”.

Estadísticas

Vídeo 1

Coches

Debido a que los coches son pequeños se ha establecido que la ventana mínima para considerar un coche sea también pequeña. De esta manera, cuando el coche está de frente o algo ladeado el clasificador es capaz de detectar 1 o 2 coches.



Matrículas

En este caso las matrículas no son capaces de ser detectadas al ser muy pequeñas y no estar bien enfocadas.

Vídeo 2

En este caso, tanto el coche como la matrícula son detectados correctamente debido a que el coche está centrado en la imagen.

Vídeo 3

En este caso, tanto el coche como la matrícula son detectados correctamente salvo en las ocasiones en las que el coche está demasiado alejado o demasiado ladeado.
También hay que hacer notar que, debido a que la ventana mínima con la que se considera si un coche es detectada o no tiene un valor bajo (para detectar coches lejanos), en ocasiones marca ciertas partes de la imagen como un coche cuando no lo son.


WA-810-TU


WA-810-TU


WA-810-TU

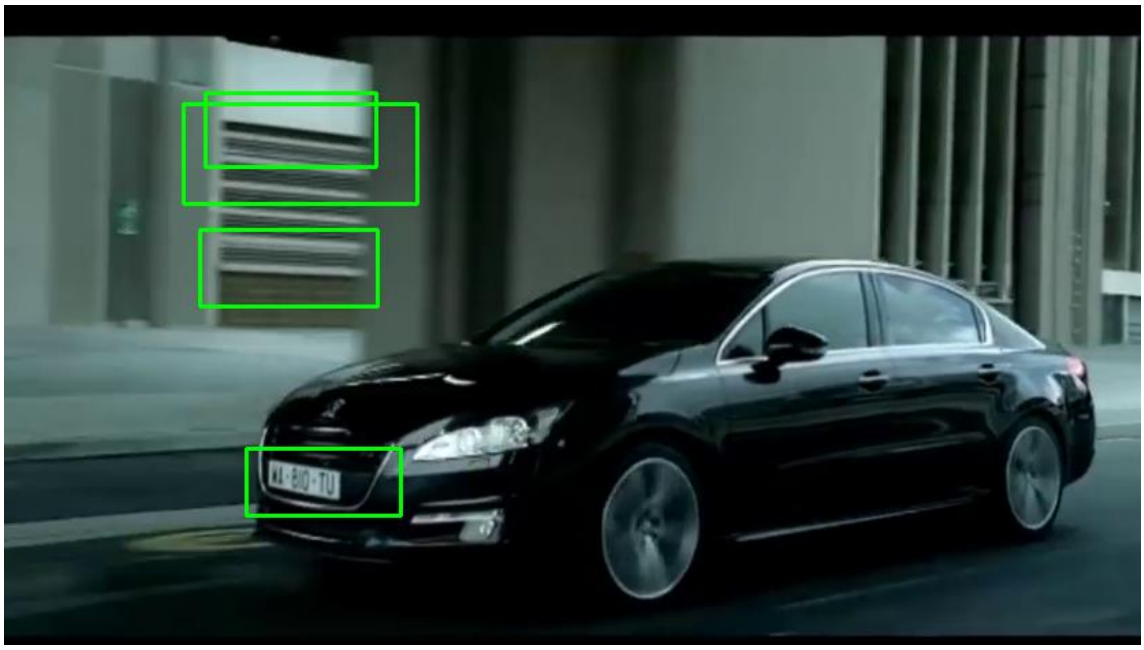


Imagen con defectos en la detección