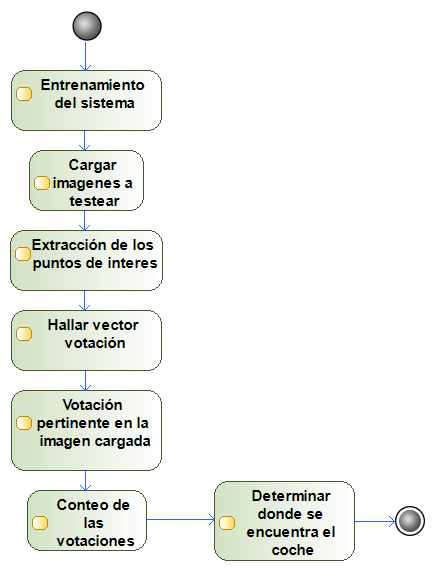
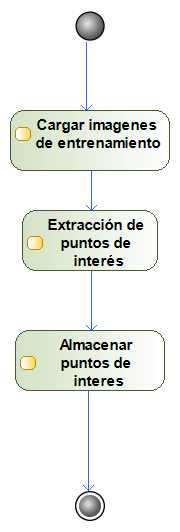
práctica 1

Agustín Daniel Schüler Allub y Patricia de Gregorio Ruiz

# Introducción

Antes de dar una explicación en detalle de la práctica nos ha parecido pertinente dar una pequeña introducción. Según el documento proporcionado para la realización de la práctica tenemos varios puntos a realizar:

* **Punto 2**. **Detección de coches mediante puntos de interés**. En este caso se nos propone la realización de un algoritmo basado en la votación de Hough. A grandes rasgos, este diagrama UML detalla los pasos a seguir para la realización del algoritmo.

**Diagrama 1. Entrenamiento del sistema.**

**Diagrama 2. Testeo de una imagen determinada.**

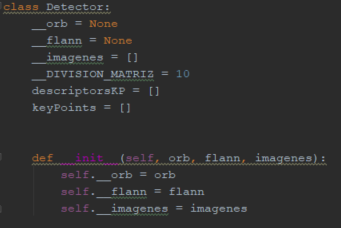
* **Punto 3 y 4**. **Detección de coches usando cv2.CascadeClassifier y detección de coches en secuencias de vídeo**. El punto 3 se basa en realizar un cierto algoritmo con ciertas funciones de OpenCV, que dado un archivo en formato XML que permite entrenar al sistema en la detección de ciertos objetos (en nuestro caso son matrículas y coches) permite pintar los rectángulos o “caras” en la imagen y decir donde se encuentra el coche y su determinada matrícula.

En el punto 4 se nos propone usar los algoritmos obtenidos en los puntos anteriores para decir donde se encuentran ciertos coches en dos videos. Se detalla pertinentemente en el apartado siguiente.

# Explicación detallada y justificación del código

Como hemos detallado en la introducción se nos ha propuesto, en el **punto 2**, la creación de un algoritmo que, dadas unas imágenes de entrenamiento, se procede a entrenar el sistema para después ponerlo a prueba con imágenes de test. En nuestro caso separamos la explicación en varios puntos:

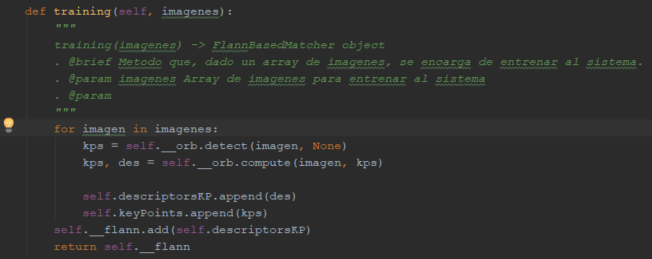
* Clase Detector.



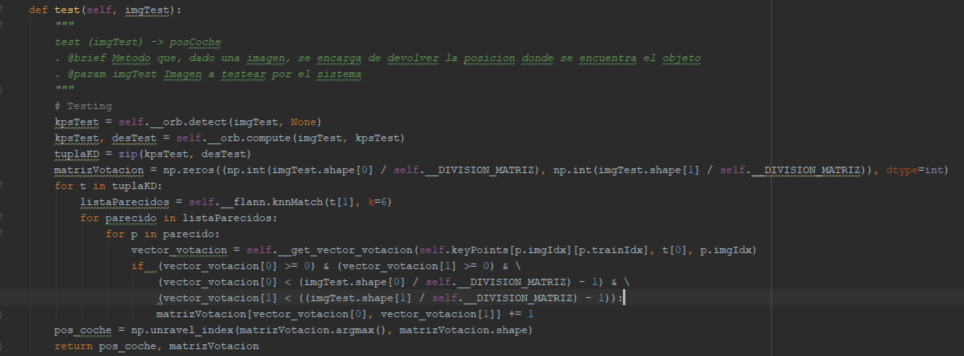
En un principio en Python podríamos ejecutar todo el código directamente en un solo fichero, pero hemos decidido separar la funcionalidad en clases para que sea mas fácil de detallar y explicar. Al crear una instancia de este objeto obtenemos un objeto capaz de ser **entrenado** y **evaluado**.

En esta primera captura tenemos el principio de la clase Detector. Tenemos varios atributos de la clase. Los dos primeros atributos son los encargados de la realización de la gran parte de la funcionalidad del algoritmo. El atributo **“orb”** lo usamos para extraer los puntos de interés y los descriptores de las imágenes. El segundo atributo **“flann”**, es una estructura de datos encargada de emparejar puntos de interés basándose en **los k vecinos mas cercanos**.

Posteriormente se procede a la realización del constructor de la clase llamado **“\_\_init\_\_”**. Cabe destacar que el atributo **“self”** que tiene el constructor y que tienen todas las clases, es un atributo especial que tienen todos los métodos de una clase que sirve para referenciar a los atributos de la propia clase. Serie el equivalente al **“this”** de Java. **“DIVISION\_MATRIZ”** es una constante que usamos para dividir o parcelar la imagen en celdas, tiene que ver con la votación, se verá en mayor detalle en los métodos pertinentes. Los atributos **“descriptoresKP”** y **“keyPoints”** son arrays encargados de almacenar los descriptores y los keypoints para usarlos posteriormente.



En esta captura tenemos el método **“training”**, este método se encarga de cargar al sistema con las imágenes de entrenamiento. Recorremos el array de imágenes, obteniendo por cada una, **los puntos de interés y los descriptores de las imágenes**, para después almacenarlos en los arrays. Al acabar de recorrer todas las imágenes, guardamos los descriptores en la estructura de datos **“flann”** y posteriormente se devuelve. En este punto, después de llamar a la **función training**, tendríamos a nuestro sistema listo para **evaluar una imagen de un coche y determinar donde se encuentra dicho coche.**

****

En este caso tenemos el método test. Dada una imagen de test, se devuelve en una tupla la posición (y, x) del coche. De forma parecida al método anterior, sacamos los descriptores y los puntos de interés de la imagen a evaluar.

Dado que necesitamos recorrer todos los descriptores y los puntos de interés, usamos la función zip de Python, que dado dos arrays, nos devuelve un array de tuplas, en cada posición hay una tupla con los valores respectivos de cada array (En la primera posición del array devuelto habría una tupla que contendría las primeras posiciones de los array dados y así sucesivamente).

Procedemos a recorrer dicho array de tuplas. Este es un punto crítico del algoritmo: Llamamos al método **“knnMatch”** de nuestro objeto **“flann”** y le pasamos los descriptores, esto sería la segunda posición de las tuplas y el valor de k. Este valor de k significa que queremos los k vecinos más cercanos a los descriptores dados. En nuestro caso serian los 6 vecinos mas cercanos.

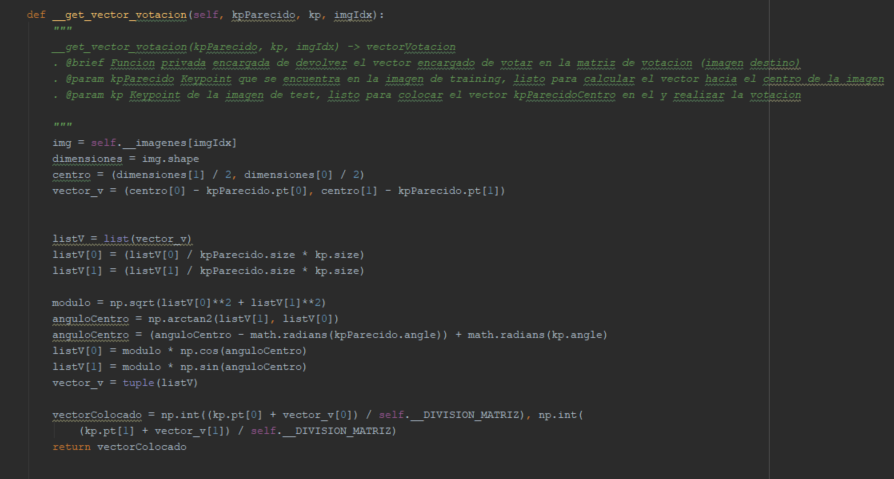
De ahí obtendríamos el objeto **“listaParecidos”** que contiene los vecinos mas cercanos del descriptor dado. Ahora bien, nos toca recorrer esos descriptores mas parecidos, dado que son los principales candidatos para realizar la votación. Pero ahí vemos 2 bucles. Esto es porque **“knnMatch”** devuelve un array de arrays, donde en cada posición se encuentran los 6 (o menos) vecinos mas cercanos.

Por cada k parecido en cada una de las posiciones de la lista devuelta por **“knnMatch”,** obtenemos el vector que va a votar en la imagen llamando a la función **“get\_vector\_votacion”** y nos va a decir donde se encuentra el coche. A esta función se le proporciona el punto de interés de la imagen de test y el punto de interés de la imagen de training, obviamente son los puntos de interés que se parecen entre ellos de cada imagen. Además también se le pasa el índice de la imagen tratada en ese preciso momento.

Ahora se realizan una serie de comprobaciones antes de realizar la votación. En primer lugar, que el vector votación devuelto positivo y en segundo lugar que no se salga de los limites de la imagen.

Una vez validadas dichas condiciones, se vota en la matriz de votación. Cabe de decir que la matriz de votación es simplemente una matriz de enteros que contiene 0’s cuyas dimensiones son las de la foto a evaluar.

Ahora bien, después de votar, sabemos que la posición del coche será la (i, j) de la matriz de mayor valor. Para ello usamos las funciones **“np.unravel\_index”** y **“argmax”**. La función **“unravel\_index”** nos devuelve la posición del valor que le pasamos como le pasamos el valor máximo usando **“argmax”, “pos\_coche”** contendrá eso precisamente: La posición del coche en la imagen lista para ser devuelta.



Por último, tenemos el método privado de la clase “get\_vector\_votacion”, es el encargado de dar a nuestro método test el vector que se encarga de votar. En primer lugar, obtenemos las dimensiones de la imagen, esta imagen es una imagen de entrenamiento.

Sacamos el vector al centro de la imagen. Necesitamos el vector al centro de la imagen porque así opera la votación a la Hough, este vector se colocará posteriormente en la imagen de test para realizar la votación, si todo va bien, la gran mayoría de los vectores votaran en la posición del coche.

Siguiendo con el flujo del código, sabemos que dos puntos de interés pueden ser iguales, pero su escala y su orientación pueden ser diferentes. Así que procedemos a adaptar el vector de votación para que la votación sea real:

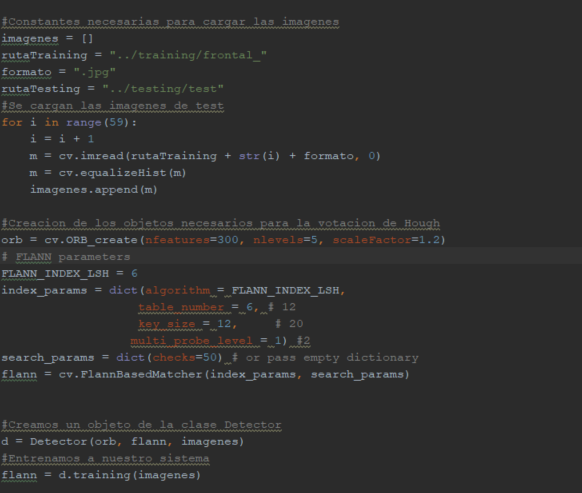
* + En el caso de la escala, lo que hacemos es “quitarle” la escala que tiene en la imagen de entrenamiento dividiendo entre ella y multiplicamos por la escala de del keypoint en la imagen de test. Así adaptaríamos la escala.
  + En el caso de la orientación, es parecido a lo anterior pero en este caso se trata de ángulos. Para ello, necesitamos pasar el vector al centro a coordenadas polares para poder sumar o restar ángulos. Hallamos su modulo y su ángulo usando **“arctan2”.** Procedemos a adaptar el ángulo, para ello se sigue la lógica que hemos usad en al escala. Al ángulo hallado, le restamos el ángulo del punto interés de la imagen de entrenamiento y le sumamos el ángulo de la imagen a evaluar. Hay que destacar que tenemos que pasar los ángulos de los puntos de interés a radianes, porque **“arctan2”** devuelve el ángulo en radianes, también valdría con pasar el ángulo devuelto por “arctan” a grados. Una vez hallado el ángulo volvemos a coordenadas cartesianas, multiplicando el modulo por el cos y el seno del nuevo ángulo y obtenemos las nuevas coordenadas del vector.

Por último devolvemos el vector de votación. Antes de ello debemos sumarlo al punto de interés que se encuentra en la imagen de test, con ello colocaríamos el vector en la imagen de test, así estaría listo para poder votar en la matriz de votación que se encuentra en el método test detallado anteriormente.

**NOTA:** Se puede apreciar que dividimos todas las matrices e incluso el vector de votación entre la constante **“DIVISION\_MATRIZ”** esto es porque si no hacemos esto, estaríamos votando a nivel de pixel, por lo que el algoritmo seria muy restrictivo a la hora de votar, así que parcelamos la imagen y votamos en esa celda de **DIVISION\_MATRIZxDIVISION\_MATRIZ.**

* Puesta a prueba del sistema de detección de coches.

Después de la realización de esta clase, procedemos a probar el sistema haciendo uso de ella.

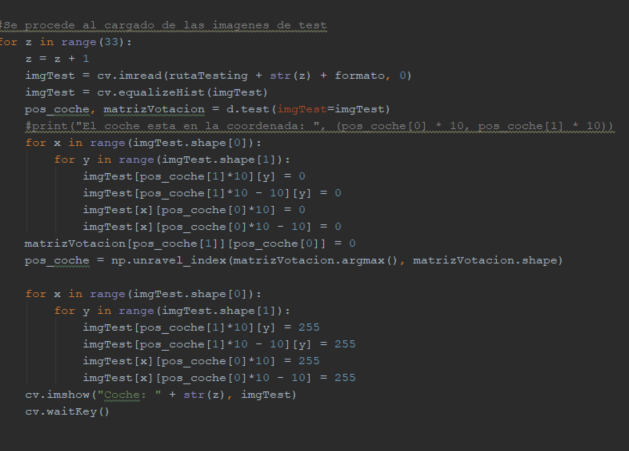


En primer lugar, cargamos nuestras imágenes de entrenamiento mediante un bucle sencillo y se almacenan en un array para poder proporcionárselo al método “training” de la clase Detector.

Ahora bien, creamos los objetos mas importantes para la detección de coches. En primer lugar, se encuentra el objeto ORB, que será el encargado de detectar puntos de interés y sus descriptores asociados. En nuestro caso, le decimos que queremos 300 puntos de interés, 5 niveles de la pirámide y un factor de escala de 1.2.

En segundo lugar, tenemos el objeto **“flann”** de la clase **FlannBasedMatcher**, que es el encargado de darnos los descriptores mas parecidos a uno dado. Lo creamos y le damos los diccionarios con los que realizará la búsqueda.

Después de todo esto, creamos un objeto de la clase Detector pasándole al constructor los objetos que necesita para la detección, para después llamar a la función training que se encargara de entrenar al sistema según las imágenes pasadas.

r

Después de la inicialización del sistema, se procede a evaluarlo.

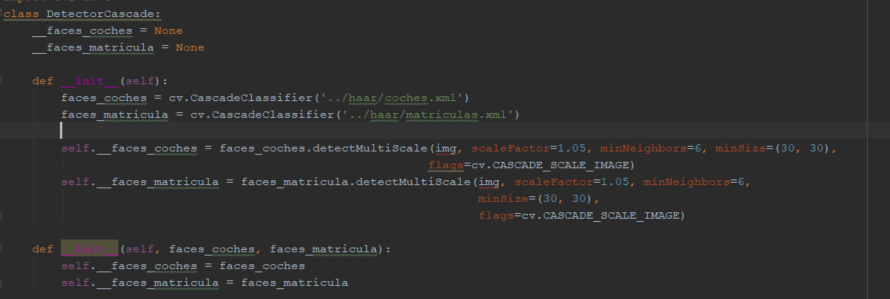
Para empezar, cargamos las imágenes a evaluar, para después pasárselas al método “test” de nuestra clase, que devolverá la posición del coche. Dada la posición del coche lo demás es trivial, dibujamos dos celdas, una negra y una blanca. Esto lo hacemos por si nuestro algoritmo es capaz de detectar más de un coche en la imagen de test, por lo que después de dibujar la primera celda, ponemos a cero el máximo y buscamos de nuevo el nuevo máximo y lo dibujamos la celda asociada a la dimensión del coche de color blanco. Finalmente mostramos la imagen para decir donde está el coche.

Ahora bien, también hay que cosas que destacar de los **puntos 3 y 4.**

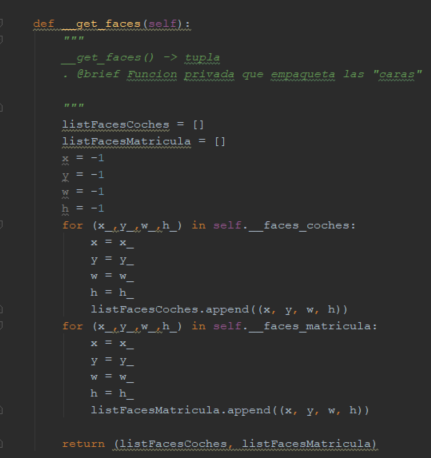
Comencemos por el **punto 3**:

* Clase Detector

En este caso también tenemos una clase Detector mucho más simple que la anterior.

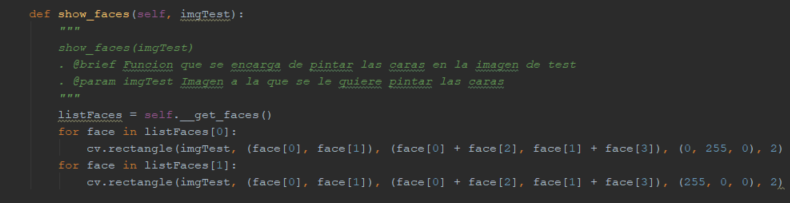


En este caso, al igual que teníamos en la clase anterior, tenemos dos constructores, uno vacío por si simplemente se quiere usar sin especificar las caras de los coches a dibujar o pasando por parámetro las caras.



Aquí tenemos el método **“get\_faces”**. Este método simplemente se encarga de empaquetar las caras en un array. Se obtiene el punto de superior izquierdo del rectángulo a dibujar, el ancho y el alto, después se añaden a un array como una tupla.

Al final se devuelve como tupla los dos arrays.



Por último, tenemos el método **“show\_faces”**, este método utiliza el método privado detallado anteriormente para dibujar los rectángulos (o caras) en la imagen pasada como parámetro.

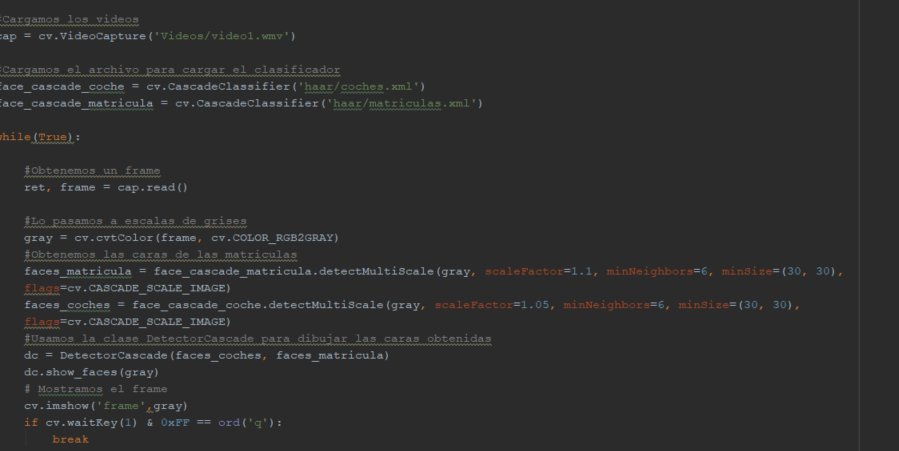
Una vez detallada la clase detector, evaluamos este sistema con las mismas imágenes de evaluación usadas en el punto 2.



Cargamos a nuestro sistema con los ficheros XML, mediante la creación de un objeto de tipo CascadeClassifier. Por cada imagen de test sacamos las caras correspondientes.

Los parámetros que se le pasan a la función **“detectMultiScale”** son: la imagen, el factor de escala, el numero mínimo de vecinos, el tamaño mínimo del objeto a detectar y las flags correspondientes. Cabe destacar que a menos factor de escala menos rendimiento a la hora de ejecutar, pero más precisión.

Creamos un objeto de tipo DetectorCascade, la clase detallada anteriormente pasándole las caras halladas, para posteriormente dibujar los rectángulos que señalaran los coches y las matrículas.



En el caso del video es idéntico que lo anterior, solo que en vez de cargar imágenes, cargamos los frames del video gracias al método **“videoCapture”** de OpenCV y en cada frame vamos dibujando los rectángulo, así en el video se vera donde se encuentra el coche a cada segundo.