Universidad de Granada

Máster Ciencia de Datos e Ingeniería de Computadores

Práctica de evaluación: Extracción de rasgos

Extracción de características

escrito por
Alberto Martín Martín

4 de marzo de 2020

Índice general

Índice de figuras

Índice de Scripts

1. Introducción

Dado que esta práctica estaba planteada para ser programada en Java, creando una aplicación con interfaz gráfica que guiase al usuario en la ejecución de las distintas partes, he querido mantener este concepto y hacer algo parecido en Python. He enfocado la práctica como una especie de aplicación CLI, de forma que se pregunta al usuario que parte de lo implementado quiere ejecutar y se le va informando en todo momento de la evolución del proceso. Quizá hubiese sido más inteligente desarrollar la práctica en un *Jupyter Notebook* ya que es posible que sea más sencillo para usted corregirla, aún así intentaré que no haya muchos problemas. La IDE que he utilizado para desarrollar la práctica es *PyCharm*, creando un entorno virtual con todos los paquetes que he utilizado. El sistema operativo en el que he trabajado es Ubuntu, y por tanto está pensada para ser ejecutada en un SO con núcleo Linux, aunque es posible que cuando esté leyendo esto la haya adaptado para ser ejecutada en Windows, dependiendo de lo que me diga hoy que tengo tutoría con usted. En la raíz del archivo comprimido en que entrego la práctica se encuentra un archivo *run.sh* mediante el cual se lanza la aplicación desde el intérprete de comandos. Este archivo tiene el siguiente contenido:

```
1 #!/bin/bash
2
3 clear
4 source venv/bin/activate
5 python3 main.py
6 deactivate
```

Script 1: Shell script que lanza la aplicación CLI en que consiste la práctica.

Lo único que hace este Script, es cargar en entorno virtual para que no haya problemas de paquetes, lanzar el archivo *main.py* y tras finalizar desactivar el entorno virtual. Si en la tutoría de hoy me dice que usted trabaja en Windows, posiblemente escriba un archivo .bat equivalente a este para ser lanzado en Windows, o algo parecido. La estructura de la práctica es la siguiente:

```
+ data
+ images_pedestrian_detection
+ lib
    - __init__.py
    - LBP.py
    - utils.py

+ venv
    - bin
    - include
    - lib
    - lib64
    - pyvenv.cfg
+ main.py
+ run.sh
+ Informe.pdf
```

El directorio *data* es el mismo que había en Prado y contiene las imágenes para entrenar al clasificador, el directorio *images_pedestrian_detection* contiene unas cuantas imágenes para testear la detección de peatones implementada, el directorio *lib* contiene

un modulo de paquetes de funciones usados en la práctica, *venv* es el directorio del entorno virtual y el resto de archivos son los que lanzan la aplicación. El archivo *LBP.py* del directorio *lib* implementa el descriptor LBP básico y uniforme que se pide en la práctica, mientras que en *utils.py* se implementan distintas funciones usadas en *main.py* para realizar tareas como leer datos o ejecutar un *K-fold*.

Para lanzar la aplicación como hemos dicho hay que ejecutar *source run.sh*, el cual activa el entorno virtual, pero para que esto funcione en su máquina, deberá cambiar el *path* del entorno virtual. Para ello, en el archivo *venv/bin/activate* cambien en la línea 40 el *path* que aparece por el suyo, donde se encuentre el directorio de la práctica en su máquina. Otra opción es ejecutar directamente en *main.py*, siempre que tenga todas las librerías instaladas no debería de haber problema. Una vez que lance la aplicación, debería de ver algo como lo que se muestra en la Figura 1.

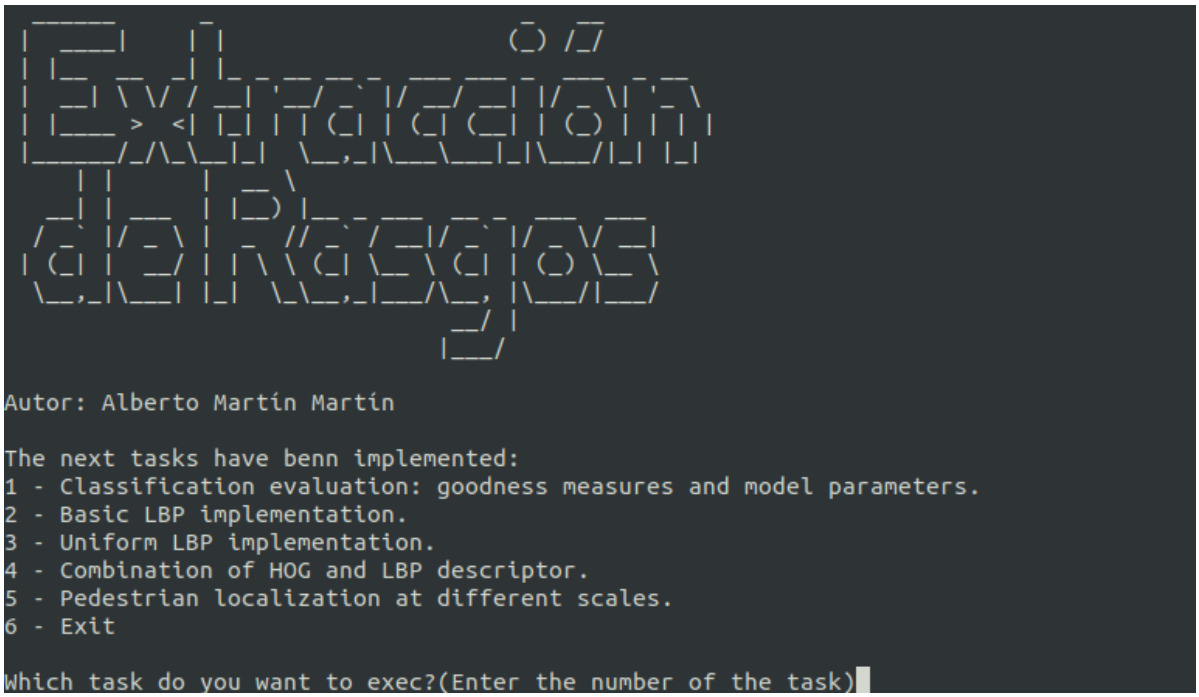

Figura 1: Incio de la aplicación

Dicho esto, comencemos con la descripción de cada una de las partes.

2. Evaluación de la clasificación: medidas de bondad y parámetros

Esta parte es la que define la estructura del resto de partes de la práctica. Lo que se pide en el guión es que juguemos un poco con los parámetros del SVM, que probemos otras particiones de *train* y *test* del *dataset* completo y que evaluemos el rendimiento del clasificador en base a distintas medidas de bondad. Yo he estructurado esta parte de forma que lo primero que se hace es leer los datos de *train* y *test* preestablecidos en el directorio *data* utilizando el descriptor HOG con los argumentos por defecto, concatenarlos, juntarlos todos y mezclándolos para formar un *dataset* completo. A continuación

se muestra el resultado de un *K-fold* implementado "a mano". Después se hace un *Grid Search* para tratar de buscar los mejores parámetros del SVM. Finálmente, se muestran los resultados sobre la división preestablecida de *train* y *test* en base al *Accuracy*, *Precision*, *Recall* y *F1 score*.

2.1. Implementación del *K-fold*

Esta parte es quizá la menos intuitiva en mi implementación, ya que he querido implementarla de forma paralela utilizando todos los núcleos del procesador y además mostrando una barra de proceso que indique al usuario en qué punto se encuentra la evaluación del *K-fold*. Para ello, he usado las librerías *multiprocessing* y *tqdm* para ambas tareas respectivamente, y he definido una clase *cross_validation* que se encarga de evaluar el rendimiento del clasificador en cada partición del conjunto de datos. Esto se hace así porque para mostrar una única barra de proceso con *tqdm.tqdm()* hay que instanciar una única vez esta función pasándole un iterable y dentro de un bucle for ir realizando las tareas que toquen. Esto es complicado de hacer cuando se quieren realizar las tareas en paralelo, y la solución que he encontrado es utilizar el método *imap* de la clase *pool* de *multiprocessing*, el cual toma como argumentos una función y un único iterable que pasa a la función argumento para ejecutar la tarea pertinente, y devuelve otro iterable correspondiente a cada tarea que se quiere realizar en paralelo. Debido a esto y a como se realiza la validación cruzada con *sklearn* era necesario definir una clase *cross_validation* que tuviese como atributos el conjunto de datos completo, el clasificador, y las distintas particiones del conjunto, y un método que tomase un solo argumento de entrada para evaluar el clasificador en la partición que indicase dicho argumento, y que devuelva los resultados. De esta forma, se le pasa este método al método *imap*, y lo que esto devuelva a la función *tqdm.tqdm()* que itera y muestra una barra de proceso marcando cuando se va completando la evaluación de cada uno de los *K-folds*.

```
Part 1.1: Own implementation of a 10-fold cross-validation with parallelization using multiprocessing pac
kage and taking all the available cores. I have manually adjusted the params of the SVM classier with C=2
, gamma=0.001 and kernel='rbf'. This combination of hyperparameters give a good result in term fo accurac
y using HOG descriptor
It takes like 4 or 5 minutes to compute, do you want to exec this?(Y/n)y

100%|████████████████████████████████████████████████████████████████| 10/10 [03:07<00:00, 18.73s/it]

Mean metrics:
|   Accuracy |   Precision |   Recall |   F1 score |
|------------+-------------+----------+------------|
|   0.955783 |    0.967762 | 0.931757 |   0.949282 |

Press ENTER to continue...
```

Figura 2: Resultados de la implementación del *K-fold* con el descriptor HOG.

Sinceramente no se si es la mejor solución, pero es la que he encontrado, y la verdad es que así explicada parece un poco liosa, así que dejo un fragmento del código que explique esta parte, el cual se muestra en el Script 2. Con el código delante y el párrafo anterior creo que se entiende bien.

En cuanto a la selección de los parámetros del modelo SVM, esta se realizó a mano, tratando de ver en que rango de estos parámetros se obtenían mejores resultados, pero los que se han seleccionado para esta parte no son los mejores, son simplemente unos con los que no se obtienen malos resultados.

```python
class cross_validation:
    def __init__(self, model, data, labels, kf):
        """
        :param model: sklearn object, it has to have a fit and predict
            method.
        :param data: numpy array, full dataset.
        :param labels: numpy array, full labels.
        :param kf: sklearn kfold object.
        """
        self.model = model
        self.data = data
        self.labels = labels
        self.train_index = []
        self.test_index = []
        for train_index, test_index in kf.split(data):
            self.train_index.append(train_index)
            self.test_index.append(test_index)
    def compute(self, i):
        """
        :param i: integer, iteration of the kfold.
        :return: float, accuracy of prediction in the ith kfold.
        """
        kf_train, kf_test = self.data[self.train_index[i], :], self.data[
            self.test_index[i], :]
        kf_train_labels, kf_test_labels = self.labels[self.train_index[i
            ]], self.labels[self.test_index[i]]
        self.model.fit(kf_train, kf_train_labels)
        prediction = self.model.predict(kf_test)
        acc = accuracy_score(kf_test_labels, prediction)
        precision = precision_score(kf_test_labels, prediction)
        recall = recall_score(kf_test_labels, prediction)
        f1 = f1_score(kf_test_labels, prediction)
        return [acc, precision, recall, f1]

C = 2
gamma = 0.001
kernel = "rbf"
svm_classifier = svm.SVC(C=C, kernel=kernel, gamma=gamma)
k = 10
kf = KFold(n_splits=k)
pool = multiprocessing.Pool(multiprocessing.cpu_count())
cross_val = cross_validation(svm_classifier, full_dataset, full_labels, kf
    )
kf_metrics = list(tqdm.tqdm(pool.imap(cross_val.compute, range(k)), total=
    k))
pool.close()
kf_metrics = np.mean(np.array(kf_metrics), axis=0)
metrics = {"Accuracy": (kf_metrics[0],),
       "Precision": (kf_metrics[1],),
       "Recall": (kf_metrics[2],),
       "F1 score": (kf_metrics[3],)}
```

Script 2: Implementación del K-fold en paralelo.

2.2. Búsqueda de los mejores parámetros del modelo SVM

Para realizar esta parte, simplemente hemos utilizado la función *GridSearchCV()* del paquete *sklearn*, la cual directamente implementa un barrido de parámetros y permite que este se realice en paralelo fijando el número de núcleos. Debido a que este proceso toma mucho tiempo, tampoco podemos hacer un barrido muy extenso, por lo que nos hemos limitado a variar el *kernel* entre lineal y *rbf*, el parámetro de coste C con los valores 1, 5 y 10, y el parámetro *gamma* con los valores 0.1, 0.01 y 0.001, dando un total de 18 combinaciones posibles y evaluándose cada una sobre un 10-fold, siendo un total de 180 veces las que hay que entrenar al modelo. Esta parte es la que más tiempo toma, pero nos sirve para darnos una idea más precisa de cuales son los mejores parámetros con este descriptor. Para valorar qué modelo es más bueno, se ha tomado como medida de bondad el *accuracy*, ya que no se pueden tener en cuenta todas, y esta es la más común de todas.

Al finalizar, se muestra una tabla con los valores adoptados para cada uno de los 18 modelos, la media y desviación típica del *accuracy* y la posición en el ranking de cada modelo, tal y como se muestra en la Figura 3.

```
[Parallel(n_jobs=8)]: Done 180 out of 180 | elapsed: 50.7min finished

Results:
|    |   C |   gamma | kernel   |   Mean accuracy |   Std accuracy |   Ranking |
|----+-----+---------+----------+-----------------+----------------+-----------|
|  9 |   5 |   0.01  | rbf      |        0.980574 |     0.00679314 |         1 |
| 15 |  10 |   0.01  | rbf      |        0.979279 |     0.00839746 |         2 |
|  3 |   1 |   0.01  | rbf      |        0.972987 |     0.0087032  |         3 |
| 17 |  10 |   0.001 | rbf      |        0.972802 |     0.00853785 |         4 |
| 11 |   5 |   0.001 | rbf      |        0.966512 |     0.00876144 |         5 |
| 14 |  10 |   0.01  | linear   |        0.963002 |     0.00899802 |         6 |
| 12 |  10 |   0.1   | linear   |        0.963002 |     0.00899802 |         6 |
| 10 |   5 |   0.001 | linear   |        0.963002 |     0.00899802 |         6 |
|  0 |   1 |   0.1   | linear   |        0.963002 |     0.00899802 |         6 |
|  6 |   5 |   0.1   | linear   |        0.963002 |     0.00899802 |         6 |
|  4 |   1 |   0.001 | linear   |        0.963002 |     0.00899802 |         6 |
|  2 |   1 |   0.01  | linear   |        0.963002 |     0.00899802 |         6 |
| 16 |  10 |   0.001 | linear   |        0.963002 |     0.00899802 |         6 |
|  8 |   5 |   0.01  | linear   |        0.963002 |     0.00899802 |         6 |
|  7 |   5 |   0.1   | rbf      |        0.957822 |     0.00891495 |        15 |
| 13 |  10 |   0.1   | rbf      |        0.957822 |     0.00891495 |        15 |
|  1 |   1 |   0.1   | rbf      |        0.955602 |     0.00849142 |        17 |
|  5 |   1 |   0.001 | rbf      |        0.944498 |     0.012391   |        18 |

Press ENTER to continue...
```

Figura 3: Barrido de parámetros SVM para el descriptor HOG.

Como podemos ver en la Figura 3, el modelo que mejores resultados proporciona es aquel que tiene un parámetro de coste igual a 5, un *gamma* igual a 0.01 y que usa un *kernel rbf*. Según el ranking, parece que los mejores resultados se obtiene usando un *kernel rbf*, y con unos valores entre 5 y 10 para el parámetro de coste, y entre 0.01 y 0.001 para *gamma*, por lo que para el entrenamiento final sobre los datos de *train* y *test* nos moveremos en torno a esos valores. Nos ha faltado por probar el *kernel* sigmoide, ya que hubiese incrementado bastante el tiempo de procesamiento y no tengo tanto tiempo. Además, los resultados al variar "a mano" los parámetros usando este *kernel* fueron bastante peores que con otros núcleos.

2.3. Evaluación del mejor clasificador sobre la partición predefinida

Para lograr los mejores resultados en esta partición debemos variar el parámetro de coste al valor 10, y el parámetro *gamma* a 0.02, obteniendo los siguientes resultados.

```
Part 1.3: Evaluate the goodness of the classifier under different type of measures like the precision, re
call, F1 score or Kappa parameter. The classifier was trained with the same hyperparameters that in the p
art 1.1
It takes like 2 or 3 minutes to compute, do you want to exec this?(Y/n)3

Training classifier...

Metrics:
|   Accuracy |   Precision |   Recall |   F1 score |
|------------+-------------+----------+------------|
|   0.979091 |    0.989733 |    0.964 |   0.976697 |

Confusion matrix:
|            |   Background |   Pedestrian |
|------------+--------------+--------------|
| Background |          595 |            5 |
| Pedestrian |           18 |          482 |

Press ENTER to continue...
```

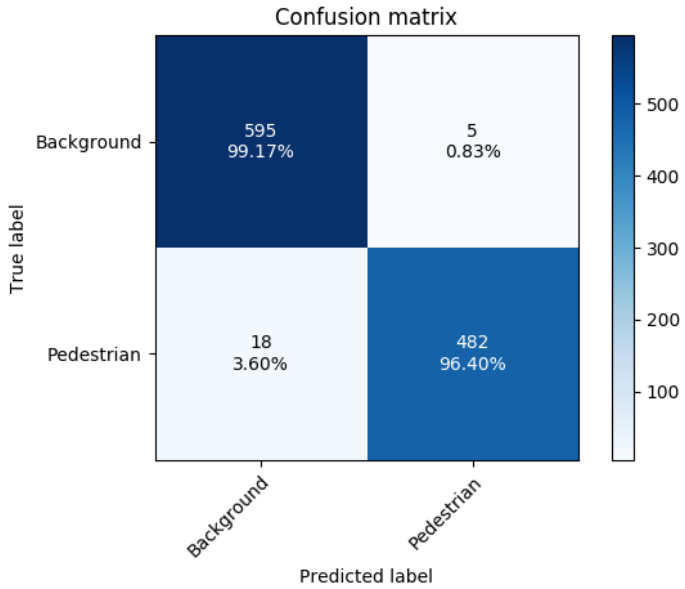

Figura 4: Resultados obtenidos con los datos de *train* y *test* preestablecidos usando el descriptor HOG.

El hecho de variar los hiperparámetros del modelo para obtener unos resultados un poco mejores en realidad lo hago a título personal, por sentirme mejor al ver unos números un poco más elevados. Sin embargo, esto no es lo que se debe hacer en un problema real, ya que esto se corresponde con una única partición del *dataset* completo, para la cual en concreto funcionan mejor estos valores de C y *gamma*. En un caso real deberíamos de quedarnos con los mejores valores mostrados en el apartado anterior, dado que en media en las diez particiones según la medida de bondad elegida el modelo funcionaba mejor. De cualquier forma, realizaremos el ejercicio de análisis con estos valores.

Empecemos con el *accuracy*. Esta es una medida muy básica pero es la típica que se utiliza a la hora de seleccionar un modelo u otro o en los concursos de ciencia de datos

a la hora de comparar resultados, lo cual es comprensible ya que simplemente nos dice cuantas veces hemos acertado con respecto al número total de casos evaluados. A pesar de esto, de cara a realizar un análisis más en profundidad es conveniente barajar otras medidas de calidad. Si por ejemplo, en vez de este problema estuviésemos tratando de detectar posibles terroristas en base a imágenes de una cámara de seguridad, el tener un *accuracy* muy elevado indicaría que el modelo funciona bien la mayoría de las veces, pero el hecho de que el modelo devuelva falsos negativos debería de tener de alguna forma más peso, ya que el modelo nos dice que un transeúnte x es una persona "normal" y lleva en realidad lleva una bomba en la maleta. No es nuestro caso, pero siempre conviene prestar atención a otras medidas de calidad. Supongamos que en lugar de *pedestrian* esta clase es *terrorist*, ¿qué nos está diciendo la medida *precision*? Pues bien, esta medida se calcula como la proporción de verdaderos positivos con respecto a la suma de verdaderos positivos y falsos positivos, en nuestro caso $\frac{482}{482+5}$. Vemos que este valor es muy elevado porque hemos cometido pocos falsos positivos, el modelo nos decía que ahí había un terrorista y por suerte no había nada, la imagen era solo fondo. Por otro lado, atendiendo ahora a la medida *recall*, la cual se calcula como la proporción de verdaderos positivos con respecto a la de verdaderos positivos y falsos negativos, es decir $\frac{482}{482+18}$, vemos que esta medida es de menor valor. Ojito, estamos diciendo que en la imagen no hay nada y tenemos a un terrorista. Entonces, ¿qué medida debería ser más importante para evaluar el modelo? Pues en el caso de los terroristas claramente el *recall*, ya que no podemos estar tan orgullosos de nuestro modelo con un 97.9 % de *accuracy* cuando estamos cometiendo 18 errores de consecuencias tan catastróficas. En el caso de los peatones, pues bueno no es tan alarmista el cometer falsos negativos, pero si la empresa que nos contrata quiere que vigilemos las inmediaciones de sus instalaciones secretas en busca de posibles espías industriales (por ejemplo), deberíamos de decirles que "oye, el modelo funciona bien la mayoría de veces, pero en contables ocasiones no detectamos a nadie cuando sí lo hay". Atendiendo al motivo de este resultado, debemos tener en cuenta que tenemos más imágenes de fondo que de peatones, por tanto es normal que el modelo clasifique más veces la imagen como fondo que como peatón. Deberíamos por tanto pedirle a la empresa si quiere mejores resultados que nos proporciones más imágenes de peatones, generarlas de alguna forma o simplemente balancear el *dataset* quitando imágenes de fondo.

Por último, ¿qué pasa si tanto los falsos negativos como los falsos positivos son motivo de alarma? Pues en ese caso recurrimos al *F1 score*. Esta medida de calidad tiene en cuenta tanto *recall* como *precision*, de forma que se calcula, suponiendo que estos dos valores son resistencias eléctricas, como dos veces la suma en paralelo de estas resistencias, es decir:

$$2 * \frac{recall * precision}{recall + precision}$$

De esta forma tenemos un valor intermedio entre las dos medidas anteriormente mencionadas. De cara a analizar yo personalmente prefiero ver *recall* y *precision* por separado, pero bueno a la hora de seleccionar un modelo, si los falsos positivos y falsos negativos son muy importantes para nosotros, convendría elegirla en lugar del *accuracy* como medida para seleccionar el modelo. También hubiese sido interesante analizar la medida *Kappa*, que de alguna forma tiene en cuenta la probabilidad de haber acertado

de casualidad, pero he leído artículos en contra de esta medida y tampoco tengo tanto tiempo, así que termino aquí.

3. Implementación del LBP básico

3.1. Explicación del código

Para que se entienda bien la implementación de este descriptor he dejado en el Script 3 el código empleado, el cual sirve tanto para el LBP básico como para el uniforme. Pues bien, la implementación se realiza mediante una clase LBP, la cual se inicializa con los valores de tamaño de ventana, tamaño de celda, desplazamiento de la celda en la ventana, número de vecinos y radio. El atributo tamaño de ventana quizá no era tan necesario, pero lo incluí debido a un error de concepto, pensaba que iba a implementar esto de una forma y luego lo hice de otra, luego lo explicaré. A pesar de esto, la clase cumple su función.

Empecemos con el método *compute_imgLBP()*, el cual nos calcula para cada píxel de la imagen el patrón correspondiente. Lo primero que hacemos es calcular cuales son los vecinos a tener en cuenta en base al radio y al número de vecinos, de forma que dividimos una circunferencia de ese radio en una cantidad de arcos igual al número de vecinos. De esta manera calculamos la posición relativa de cada vecino con respecto al píxel de referencia, es decir, el desplazamiento de un píxel en ambos ejes para que coincida con cada vecino, lo cual se corresponde con las variables *nn_x_shift* y *nn_y_shift*. Una vez hacemos esto, podríamos recorrer cada píxel, ver cuales son sus vecinos, obtener el patrón decimal y pasarlo a decimal. Sin embargo, para que esto sea más eficiente, podríamos hacer la comparación de de todos los píxel de la imagen iterativamente con cada uno de sus vecinos de una, desplazando la imagen lo que se corresponda para cada vecino y comparando la imagen completa con la desplazada, multiplicando además por la potencia de dos con la que se corresponda esa comparación. De esta forma, se obtiene mucho más rápido el patrón de cada píxel. También tengo que decir que esto no se me ocurrió a mi, se le ocurrió a Francisco Luque, compañero de clase, quién me lo contó y cambié mi código.

El método *compute()* llama a este para finalmente calcular el descriptor de la imagen. Aquí es donde está el pequeño error de concepto. Yo inicialmente pensaba que iba a usar este método para detección de peatones en imágenes completas, de forma que pasaríamos una imagen grande con muchos peatones, moveríamos una ventana sobre ella, y para cada ventana calcularíamos un descriptor en base al histograma de patrones para el desplazamiento de celdas, es por eso que hay un bucle for para mover la ventana. Esto no se hace así, la ventana se mueve en un algoritmo superior a este, y se llama al método de esta clase para calcular el descriptor de cada ventana, por lo que es tontería definir un atributo interno a la clase con el tamaño de la ventana. Si pasamos una imagen de tamaño igual a la venta definida al instanciar la clase, se devuelve un único descriptor, por lo que a pesar de este pequeño error esto funciona. A parte de esto no hace falta comentar mucho más, se calcula el histograma por celdas, se normaliza y se concatena, no tiene más.

.

```
class LBP:
    def __init__(self, window_size=(128, 64), cell_size=(16, 16), delta
        =(8, 8), num_nn=8, radius=1):
        """
        :param window_size: tuple. Size of the window that move around the
            full image. The full image size has to be
        multiple of the window size.
        :param cell_size: tuble. Size of the cell that move around the
            window. The window size has to be multiple of
        the cell size.
        :param delta: tuple. Number of pixel that the cell size moves in
            each iteration.
        :param num_nn: integer. Number of nearest neighbour to compute.
        :param radius: integer. Distance from neighbour to the pixel.
        """
        self.window_size = window_size
        self.cell_size = cell_size
        self.delta = delta
        self.num_nn = num_nn
        self.radius = radius
        # Generating Look up table for uniform LBP descriptor.
        lut = np.zeros(2 ** self.num_nn)
        E = 0
        for n in range(len(lut)):
            pattern = bin(n)[2:].zfill(self.num_nn)
            U = 0
            bit = pattern[-1]
            for i in range(len(pattern)):
                if bit != pattern[i]:
                    U += 1
                    bit = pattern[i]
            if U > 2:
                lut[n] = -np.inf
            else:
                lut[n] = E
                E += 1
        lut[np.where(lut == -np.inf)[0]] = E
        self.lut = lut

    def compute(self, img, method="basic"):
        """
        :param img: numpy.array. Full image.
        :param method: string, Type of lbp, "basic" or "uniform".
        :return: descriptor of the image.
        """
        imgLBP = self.compute_imgLBP(img, method)
        img_x_size, img_y_size = img.shape
        LBPdescriptor = []
        for window_x, window_y in product(range(0, img_x_size, self.
            window_size[0]),
                                          range(0, img_y_size, self.
                                              window_size[1])):
            windowLBPdescriptor = []
            window = imgLBP[window_x:(window_x + self.window_size[0]),
```

```
                window_y:(window_y + self.window_size[1])]
49              for cell_x, cell_y in product(range(0, self.window_size[0] -
                    self.delta[0], self.delta[0]),
50                                            range(0, self.window_size[1] -
                                                  self.delta[1], self.delta[1])
                                                  ):
51                  cell = window[cell_x:(cell_x + self.cell_size[0]), cell_y
                        :(cell_y + self.cell_size[1])]
52                  cellLBPdescriptor = np.histogram(cell, bins=2 ** self.
                        num_nn)[0]
53                  cellLBPdescriptor = cellLBPdescriptor/np.linalg.norm(
                        cellLBPdescriptor)
54                  windowLBPdescriptor.append(cellLBPdescriptor)
55              windowLBPdescriptor = np.array(windowLBPdescriptor).reshape
                    (-1, )
56              LBPdescriptor.append(windowLBPdescriptor)
57          return np.array(LBPdescriptor).squeeze()
58
59      def compute_imgLBP(self, img, method="basic"):
60          """
61          :param img: numpy.array. Full image.
62          :param method: string, Type of lbp, "basic" or "uniform".
63          :return: image of the same size that "img", but each pixel
                contains the decimal number of the local binary
64          pattern.
65          """
66          img_x_size, img_y_size = img.shape
67          nn_x_shift = np.round(-self.radius * np.sin(np.arange(0, 2 * np.pi
                , 2 * np.pi / self.num_nn))).astype(int)
68          nn_y_shift = np.round(-self.radius * np.cos(np.arange(0, 2 * np.pi
                , 2 * np.pi / self.num_nn))).astype(int)
69          power = np.flip(2 ** np.arange(self.num_nn))
70          imgLBP = np.zeros((img_x_size, img_y_size, self.num_nn), dtype=int
                )
71          for i in range(self.num_nn):
72              imgLBP[:, :, i] = (np.roll(img, [nn_x_shift[i], nn_y_shift[i
                    ]], axis=(0, 1)) >= img) * power[i]
73          imgLBP = np.sum(imgLBP, 2)
74
75          if method == "uniform":
76              # Map patterns to uniform patterns
77              imgLBP = self.lut[imgLBP]
78
79          return imgLBP.astype(int)
```

Script 3: Implementación del descriptor LBP

3.2. Resultados obtenidos

Los resultados obtenidos con este descriptor son aproximadamente iguales a los que se obtienen con el descriptor HOG, siendo superiores para este descriptor. Como aspecto positivo, no solo obtenemos mejores resultados en *accuracy* sino también en *recall*, y ya mencionamos anteriormente la importancia que cometer falsos negativos podría tener. Como aspecto negativo, el consumo de recursos. Con una ventana de $64x128$, una celda

de 16x16 y un desplazamiento de 8 píxeles arriba y abajo para las celdas, tenemos un total de 26880 *features* frente a las 3780 que teníamos con el descriptor HOG, lo cual incrementa considerablemente el tiempo que requiere entrenar a un modelo SVM. De hecho, yo no puedo ejecutar un *K-fold* en paralelo porque se desbordan mis 8GB de RAM, tuve que ejecutarlo en una maquina virtual más potente de Azure. Yéndonos al extremo, habría que sopesar que nos importa más, si obtener unos resultados un poco mejores, o si tener que pagar un servicio de computación para entrenar nuestros modelos. En mi caso, tengo crédito Azure de estudiante pues esto no es un problema, mientras dure prefiero obtener mejores resultados, incluso trataría de mejorar los resultados si tuviese más tiempo.

```
Part 2.1: Own implementation of a 10-fold cross-validation with parallelization
using multiprocessing package and taking all the available cores. I have manuall
y adjusted the params of the SVM classifier with C=2, gamma=0.001 and kernel='rb
f'. This combination of hyperparameters give a good result in term fo accuracy u
sing the basic LBP descriptor
It takes like 9 or 10 minutes to compute, do you want to exec this?(Y/n)y

100%|████████████████████████████████████| 10/10 [20:32<00:00, 123.26s/it]

Mean metrics:
|   Accuracy |   Precision |   Recall |   F1 score |
|------------+-------------+----------+------------|
|   0.982057 |    0.979417 | 0.980791 |   0.980078 |

Press ENTER to continue...
```

Figura 5: Resultados de la implementación del *K-fold* con el descriptor LBP básico.

```
Results:
|   C |   gamma | kernel   |   Mean accuracy |   Std accuracy |   Ranking |
|-----+---------+----------+-----------------+----------------+-----------|
|  10 |   0.01  | rbf      |        0.992046 |     0.00370501 |         1 |
|   5 |   0.01  | rbf      |        0.992046 |     0.00370501 |         1 |
|   1 |   0.1   | linear   |        0.989642 |     0.00414768 |         3 |
|  10 |   0.01  | linear   |        0.989642 |     0.00414768 |         3 |
|  10 |   0.1   | linear   |        0.989642 |     0.00414768 |         3 |
|   5 |   0.001 | linear   |        0.989642 |     0.00414768 |         3 |
|  10 |   0.001 | linear   |        0.989642 |     0.00414768 |         3 |
|   5 |   0.01  | linear   |        0.989642 |     0.00414768 |         3 |
|   5 |   0.1   | linear   |        0.989642 |     0.00414768 |         3 |
|   1 |   0.001 | linear   |        0.989642 |     0.00414768 |         3 |
|   1 |   0.01  | linear   |        0.989642 |     0.00414768 |         3 |
|  10 |   0.001 | rbf      |        0.988531 |     0.00451384 |        12 |
|   1 |   0.01  | rbf      |        0.987975 |     0.00462588 |        13 |
|   5 |   0.001 | rbf      |        0.986495 |     0.00567621 |        14 |
|   1 |   0.001 | rbf      |        0.976323 |     0.00394334 |        15 |
|  10 |   0.1   | rbf      |        0.952274 |     0.0106606  |        16 |
|   5 |   0.1   | rbf      |        0.952274 |     0.0106606  |        16 |
|   1 |   0.1   | rbf      |        0.949315 |     0.0107047  |        18 |

Press ENTER to continue...
```

Figura 6: Barrido de parámetros SVM para el descriptor LBP básico.

Pues bien, en la Figura 5 vemos los resultados obtenidos con este descriptor, con los mismos hiperparámetros del primer *K-fold* mostrado en el apartado anterior. En cuanto al barrido de parámetros, este se ejecutó con la misma combinación de valores que en el caso anterior, y los resultados se muestran en la Figura 6. Como podemos ver, los resultados en cuanto a *accuracy* son en general mejores que con el descriptor HOG para todos los modelos, e incluso la desviación típica es menor, lo cual nos está diciendo que hay menor variación en los resultados para las distintas particiones. Esto nos dice que este descriptor funciona mejor en general en este problema particular, por lo que es un buen candidato a ser usado para un detector de peatones.

```
Part 2.3: Evaluate the goodness of the classifier under different type of measur
es like the precision, recall or F1 score. The classifier was trained with the s
ame hyperparameters that in the part 2.1
It takes like 2 or 3 minutes to compute, do you want to exec this?(Y/n)y

Training classifier with C = 10 gamma =  0.01 and kernel =  rbf ...

Metrics:
|   Accuracy |   Precision |   Recall |   F1 score |
|------------+-------------+----------+------------|
|   0.989091 |    0.991935 |    0.984 |   0.987952 |

Confusion matrix:
|            |   Background |   Pedestrian |
|------------+--------------+--------------|
| Background |          596 |            4 |
| Pedestrian |            8 |          492 |

Press ENTER to continue...
```

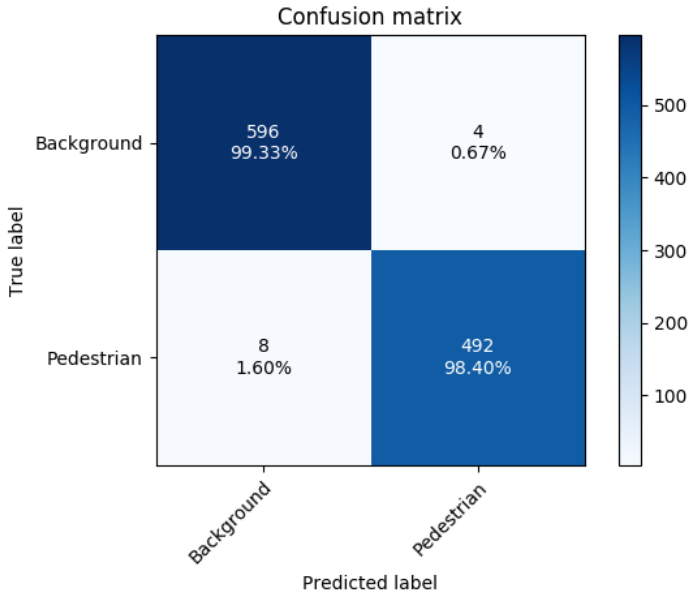

Figura 7: Resultados obtenidos con los datos de *train* y *test* preestablecidos usando el descriptor LBP básico.

Por último, en la Figura 7 se muestra el resultado final con las particiones preestablecidas para el modelo ganador del ranking anteriormente mostrado. Vemos que en

general todas las medidas han mejorado, pero lo que voy a resaltar es que ha mejorado el *recall*. Tras todo lo explicado anteriormente y considerando la importancia que podría tener cometer falsos negativos creo que es algo a remarcar el hecho de que esta medida haya mejorado. Es cierto que ajustar el modelo SVM ha sido más costoso en tiempo y en recursos que con el descriptor HOG, pero una vez calculados los vectores soporte y la región de decisión, la predicción es básicamente ver en qué lado del hiperplano se encuentra el nuevo dato, por lo que no debe de haber mucha diferencia a la hora del tiempo de procesamiento en predicción al usar un clasificador u otro. Pero también es cierto que para clasificar un nuevo dato hay que calcular su descriptor, y el descriptor HOG de Open CV tarda menos en calcularse que el descriptor LBP que he implementado yo. Poniéndonos en la situación de que pensamos usar el clasificador para procesar un video *frame* a *frame* en busca de peatones, no sabría bien qué descriptor elegir. De todas formas, en este problema en el que el tiempo no importa, me quedaría con el modelo entrenado a partir del descriptor LBP, me gusta esa mejora aunque sea pequeña.

4. Implementación del descriptor LBP uniforme

4.1. Explicación del código

A parte de lo ya comentado en la sección anterior, no hay mucho más que decir a cerca de la implementación. En el Script 3 se muestra como en al instanciar un objeto de clase LBP se genera en base al número de vecinos una *Look up table* para mapear los dígitos correspondientes al patrón binario local de cada bit a su correspondiente dígito según el método uniforme. De esta forma, todo el proceso es el mismo, pero la imagen LBP se devuelve con este mapeo aplicado.

4.2. Resultados obtenidos

Para esta parte, aunque está implementada, no se ha realizado ni el *K-fold* inicial ni el barrido de parámetros, ya que tarda demasiado tiempo y no merece la pena, teniendo en cuenta que ya lo hemos realizado con el LBP básico y que este descriptor no difiere tanto del uniforme. Directamente se muestran los resultados finales con los mismos hiperparámetros que antes.

En la Figura 9 vemos que los resultados obtenidos usando este clasificador varía poco con respecto al LBP uniforme, siendo estos un poco peores. Esto es un aspecto positivo, ya que quiere decir que el descriptor funciona de forma parecida y en cambio realiza una reducción dimensional mucho más grande que en el caso anterior, con 6195 *features* en lugar de 26880. Esto último va a repercutir en menores tiempos de entrenamiento, lo cual nos facilita un ajuste más rápido del modelo. A parte de esto no tengo mucho más que decir. El *kernel* que mejor funciona sigue siendo el rbf, por lo que parece que esta es una buena solución en este problema. Se que los resultados pueden mejorar más, pero sería cuestión de trabajar esto más.

```
Part 3.3: Evaluate the goodness of the classifier under different type of measur
es like the precision, recall or F1 score. The classifier was trained with the s
ame hyperparameters that in the part 3.1
It takes like 2 or 3 minutes to compute, do you want to exec this?(Y/n)y

Training classifier with C = 10 gamma =  0.01 and kernel =  rbf ...

Metrics:
|   Accuracy |   Precision |   Recall |   F1 score |
|------------+-------------+----------+------------|
|   0.987273 |    0.989919 |    0.982 |   0.985944 |

Confusion matrix:
|            |   Background |   Pedestrian |
|------------+--------------+--------------|
| Background |          595 |            5 |
| Pedestrian |            9 |          491 |

Press ENTER to continue...
```

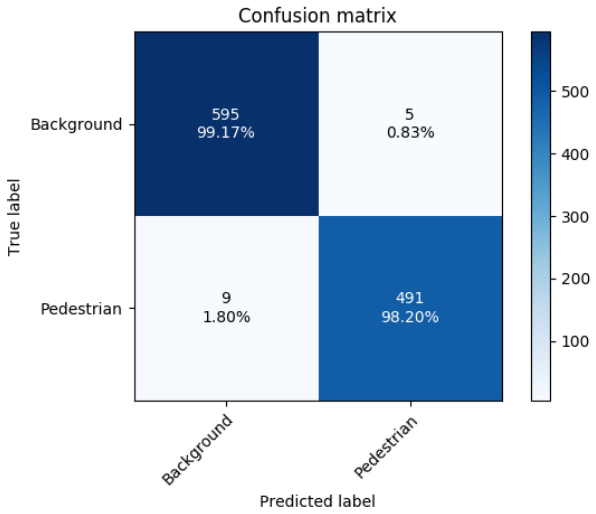

Figura 8: Resultados obtenidos con los datos de *train* y *test* preestablecidos usando el descriptor LBP uniforme.

5. Combinación de características

Por último en esta práctica referente a descriptores, se pide que probemos a entrenar un modelo usando los descriptores LBP y HOG combinados, simplemente concatenando uno detrás de otro. Yo la verdad es que esperaba que los resultados empeorasen, además lo estuve hablando contigo en tutoría, pero curiosamente mejoran un poco. Esperaba que empeorasen porque entendía que ambos descriptores al final iban a aportar información similar referente a los contornos de la imagen, a la textura, y por tanto el combinarlos de esa forma al final iba a resultar en que los datos iban a estar descritos de forma redundante, no aportando información nueva y estorbando a la hora de clasificar. Parece que no son tan redundantes como cabría esperar y se tarda más o menos lo mismo en entrenar que con el LBP básico. Tampoco puedo decir mucho más a parte de mi asombro ya que no he hecho más pruebas.

```
Part 4.3: Evaluate the goodness of the classifier under different type of measur
es like the precision, recall or F1 score. The classifier was trained with the s
ame hyperparameters that in the part 3.1
It takes like 2 or 3 minutes to compute, do you want to exec this?(Y/n)y

Training classifier with C = 10 gamma =  0.01 and kernel =  rbf ...

Metrics:
|   Accuracy |   Precision |   Recall |   F1 score |
|------------+-------------+----------+------------|
|   0.991818 |    0.997972 |    0.984 |   0.990937 |

Confusion matrix:
|            |   Background |   Pedestrian |
|------------+--------------+--------------|
| Background |          599 |            1 |
| Pedestrian |            8 |          492 |

Press ENTER to continue...
```

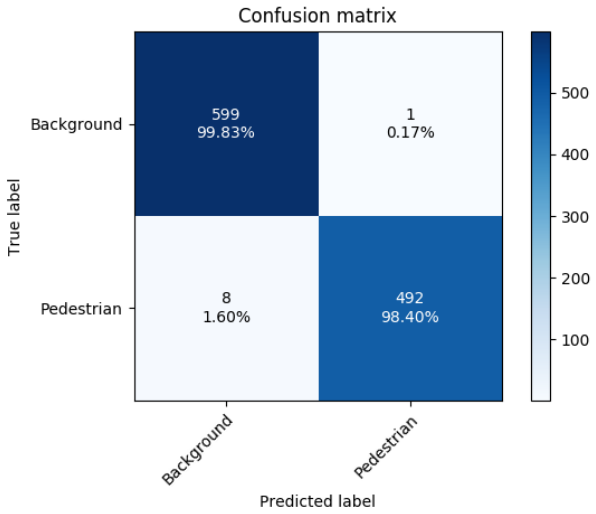

Figura 9: Resultados obtenidos con los datos de *train* y *test* preestablecidos usando los descriptores HOG y LBP uniforme combinados.

6. Detección de peatones a diferentes escalas

Esta parte por ser la última es a la que menos tiempo he tenido de dedicarle. Es por esto que no he podido implementarla como me hubiese gustado, "a mano", utilizando los clasificadores y descriptores que hemos estado viendo en los apartados anteriores. Los resultados obtenidos son bastante mediocres, pero bueno me conformaré con haber llegado a esta parte aunque lo obtenido no sea del todo satisfactorio.

Para completar esta parte de la práctica he usado un detector de peatones que viene ya implementado en Open CV, inicializando un objeto de clase HOG tal y como hicimos en la primera parte de la práctica. A este objeto se le puede asignar como atributo un clasificador SVM, el cual tiene que ser propio de Open CV, entrenado para clasificar *background* y *pedestrian*. Open CV incorpora un clasificador ya entrenado para realizar esta tarea, de forma que asignadoselo al objeto de clase HOG podemos llamar al método

detectMultiScale el cual recorre la imagen en busca de peatones. Como argumentos de entrada toma la imagen, el tamaño de ventana, el desplazamiento de la ventana, y la escala a la que queremos aplicar el detector, y como salida nos devuelve un rectángulo que marca la ventan donde se detectó al peatón. Con esto y un bucle for podremos tratar de buscar peatones a distintas escalas, y después solo quedará eliminar los rectángulos que se superponen demasiado, debido a que estos hacen referencia a la misma persona a distintas escalas. Esta última tarea la realizamos con el paquete *imutils*, que trae una función específica para este propósito. El problema es que el clasificador falla a veces, y sobre todo que no tengo acceso a la implementación del detector ni del clasificador, lo que dificulta un buen ajuste del método. Lo que hubiese hecho de disponer de más tiempo sería hacer lo mismo implementado por mí con un clasificador LBP que parece que da mejores resultados, así tendría más control de lo que sucede.

Resumiendo, en el Script 4 se muestra el código empleado para esta tarea. Una cosa que no he mencionado antes, es que para que se aplique el detector de igual forma, escalamos previamente todas las imágenes para que tengan el mismo tamaño, como se puede ver de la línea 10 a la 13. Además, aunque no aparece ahí, también estuve jugando con los mapas de colores, ya que es algo que remarcaste mucho en clase, pero tampoco conseguí mejorar el resultado ni entender bien porqué no mejoraba, así que lo excluyo del análisis.

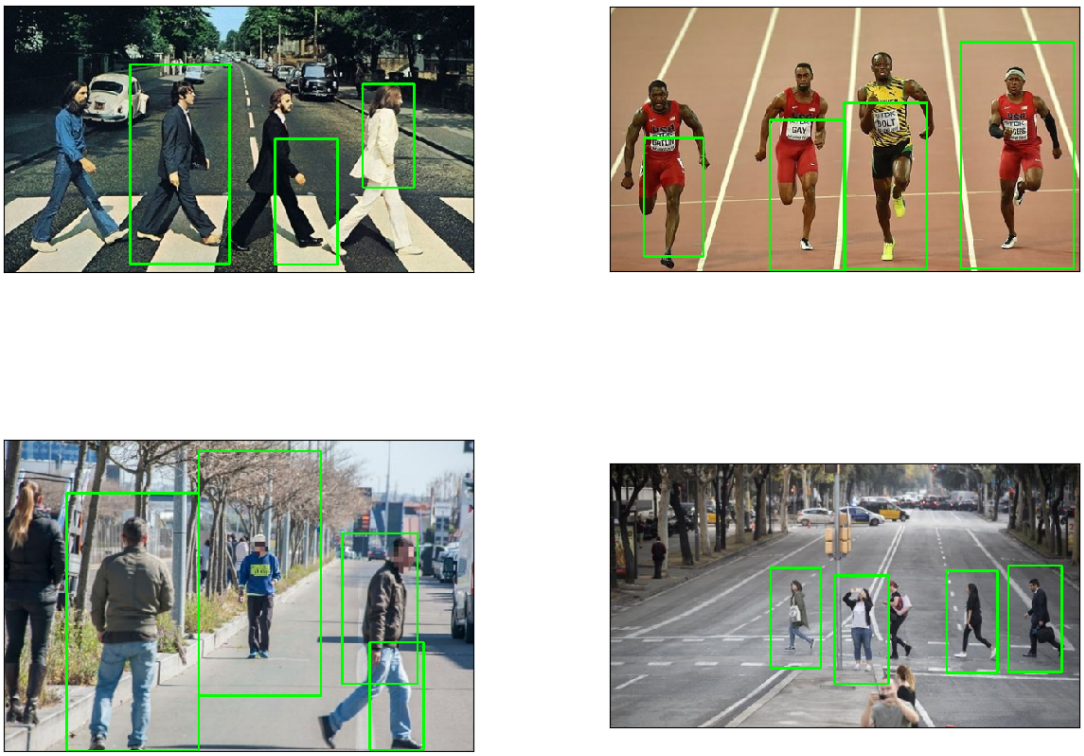

Figura 10: Resultados obtenidos en la detección de peatones.

En la figura 10 se muestran los resultados. Como podemos ver, en una imagen donde los peatones se ven tan claramente como la de los *Beatles*, no los detectamos a todos, cosa que no llego a comprender porque. He probado con diferentes escalas y con diferentes tamaños de ventana y no hay manera. Debería de haberme puesto antes para poder tener tiempo de analizar esto con un poco más detalle, pero bueno, no está mal del todo, espero que la lectura haya sido amena.

```
path = "images_pedestrian_detection/"
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

scales = [1.05, 1.25, 1.5]
for im_name in os.listdir(path):
    imgPath = path + im_name

    # Resize all images to same size
    img = cv2.imread(imgPath)
    res_width = min(600, img.shape[1])
    res_heigth = int(img.shape[0] * (res_width / img.shape[1]))
    img = cv2.resize(img, dsize=(res_width, res_heigth))
    # Detect pedestrian at diferent scale
    rects_list = []
    weights_list = []
    for scale in scales:
        (rects, weights) = hog.detectMultiScale(img,
                                                winStride=(3, 3),
                                                padding=(6, 6),
                                                scale=scale)
        rects_list.append(rects)
        weights_list.append(weights)

    # Detect overlap
    total_rects = np.array([])
    for rects in rects_list:
        rects = np.array([[x, y, x + w, y + h] for (x, y, w, h) in rects])
        total_rects = np.append(total_rects, rects)
    if total_rects.shape[0] != 0:
        total_rects = total_rects.reshape(-1, 4)
        pick = non_max_suppression(total_rects, probs=None, overlapThresh
            =0.2)
    else:
        pick = []

    # Point pedestrian on images
    for (xA, yA, xB, yB) in pick:
        cv2.rectangle(img, (xA, yA), (xB, yB), (0, 255, 0), 2)
    print(pick.shape[0], "pedestrian detected in", im_name)
    fig = plt.figure()
    plt.xticks([])
    plt.yticks([])
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    plt.show()
```

Script 4: Implementación del detector de peatones con descriptor HOG disponible en Open CV.