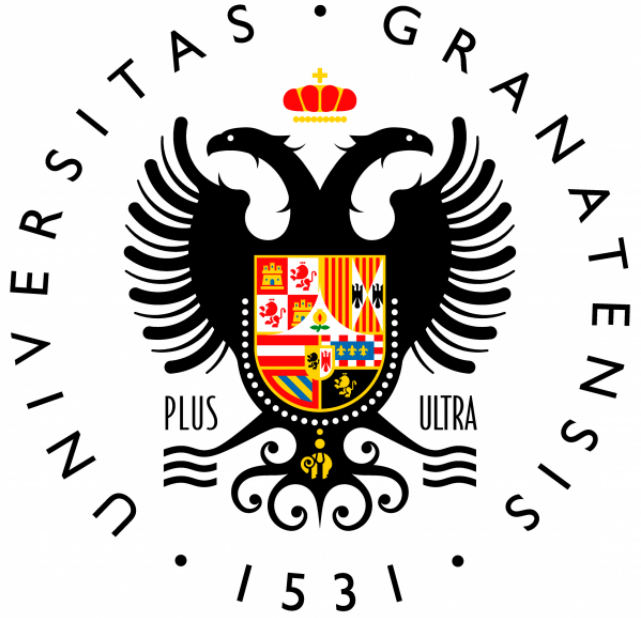
Visión Por Computador (2017-2018)
Grado en Ingeniería Informática
Universidad de Granada

Detección de peatones

Antonio Manuel Milán Jiménez
Germán González Almagro

15 de enero de 2018



Índice

Índice de figuras

1. Descripción y enfoque del problema

El problema de la detección de peatones consiste en obtener un modelo que, dada una imagen, sea capaz de detectar la presencia de un peatón en la misma. La complejidad del problema reside en la variedad de posiciones, poses y apariencias en las que podemos encontrar un peatón, por ello, será necesario obtener un método que sea capaz de extraer un conjunto de características que permita la identificación bajo condiciones de iluminación y fondo difíciles. Una vez disponemos de las características, entrenaremos un clasificador de manera que, tras el entrenamiento, este será capaz de detectar peatones en nuevas imágenes.

Para la extracción de características emplearemos el algoritmo del descriptor HOG (Histogram of Oriented Gradients). Este método se basa en la idea de que la apariencia y forma de un objeto pueden ser caracterizada mediante la información contenida en la distribución de la intensidad de los gradientes y la frontera que lo delimita, por ello se opta por utilizar este algoritmo. En la práctica, de forma similar a SIFT, consiste en dividir la imagen en pequeñas regiones, llamadas celdas, para cada una de las cuales se obtiene el histograma de las direcciones u orientación de los bordes de los gradientes sobre los píxeles contenidos en ella. Sin embargo, las características extraídas se ven altamente afectadas por condiciones de iluminación y fondo, para suavizar este efecto las normalizamos teniendo en cuenta no solo la información contenida en la propia celda, también la contenida en las celdas vecinas, a estos conjuntos de celdas los denominamos bloques. Una vez obtenidas las características, debemos entrenar un clasificador que, en base a ellas, sea capaz de predecir la presencia de peatones en imágenes. El modelo empleado para ello será una máquina de vectores de soporte (SVM).

La ventaja de este enfoque reside en que las estructuras que emplea para representar la información se elaboran en base a los gradientes, que representan de forma característica la información local. Esto, sumado a una fuerte normalización, hace al método robusto ante cambios en la fotometría, así como a transformaciones geométricas relacionadas con la orientación, pose y forma del objeto a reconocer. De esta forma, el método será capaz de identificar peatones permitiendo libertad de movimiento en las extremidades, aunque sí exigiendo una pose erguida.

2. Conjunto de datos y metodología

La base de datos empleada será INRIA-Person, que contiene numerosas imágenes tanto de personas en diferentes poses y posiciones (conjunto positivo) como imágenes donde no aparece ninguna persona (conjunto negativo).

Respecto al conjunto positivo, emplearemos 1208 imágenes de entrenamiento y 566 imágenes de test, en ambos casos junto a las imágenes espejadas asociadas, lo que da un total de 2416 y 1132 imágenes respectivamente. Sobre su tamaño, originalmente de 96x160, se han reducido a un tamaño de 64x128, que será el tamaño de nuestra ventana.

Como conjunto de entrenamiento negativo emplearemos 1218 y 453 imágenes para entrenamiento y test respectivamente de diferentes tamaños. Con el fin de obtener imágenes del tamaño deseado, sin que el sub-muestreo suponga un sesgo en el conjunto de datos, obtendremos 10 muestras de tamaño 64x128 de cada imagen de forma aleatoria. Por lo tanto, realmente tendremos 12180 imágenes negativas de entrenamiento y 4530 imágenes negativas de test.

Gracias a que en nuestro conjunto positivo las personas aparecen en diferentes poses y posiciones, la base de datos no presenta sesgo respecto a estas características, tal y como sería este problema llevándolo a la práctica.

Aun así hay que destacar que las personas aparecen erguidas en la gran mayoría de las ocasiones, esto podría suponer un sesgo en caso de que el objetivo fuera reconocer personas de forma general. Sin embargo, los peatones aparecen erguidos, por tanto el conjunto de datos resulta apropiado para entrenar un clasificador que los reconozca.

3. Implementación del método

Una vez definido el método de aproximación al problema y cargados los datos, implementamos los procedimientos necesarios para llevar a cabo los pasos definidos por el método. A continuación se detallan las cuestiones de implementación, así como las especificación de parámetros utilizada para la obtención de resultados.

3.1. Especificación de parámetros del algoritmo

La especificación de parámetros viene dada por los ideólogos del método, Navneet Dalal y Bill Triggs, que detallan en su publicación "Histogram of Oriented Gradients for Human Detection" un estudio experimental a partir del cual obtuvieron los parámetros óptimos para obtener los mejores resultados.

Emplearemos una ventana de tamaño 64x128 para el análisis, además de un tamaño de celda 6x6 y un tamaño de bloque de 3x3. De este modo, obtendremos 11x22 = 242 celdas para cada imagen, y, teniendo en cuenta que para obtener los bloques se desplaza la ventana de 3x3 celdas a modo de máscara sobre la matriz de celdas, obtenemos un total de (11-2)*(22-2) = 180 bloques, cada uno formado por 9 celdas, y donde cada celda contiene un histograma de 9 orientaciones posibles. Así pues, cada imagen vendrá descrita por un vector de 180x9x9 = 14580 características.

3.2. Aplicación de gamma-correction

La corrección gamma es una operación aplicada a las imágenes para corregir el brillo, en su forma más sencilla viene descrita por la siguiente fórmula:

$$V_{out} = A \times V_{in}^{\gamma}$$

Donde A es una constante en nuestro caso $A = 1$, y γ es el factor de corrección, en nuestro caso $\gamma < 1$, por tanto aplicamos un gamma de codificación, señalado por los autores como óptimo para obtener mejoras en los resultados. A continuación se muestra la función que aplica esta corrección a las imágenes:

"."

```
#Función que aplica la corrección gamma a una lista de
#imagenes dada como argumento
def gamma_correction(img_list, gamma):
    #Elevamos cada imagen al parámetro gamma
    return [np.power(i, gamma, dtype = np.float32) for i in img_list]
```

3.3. Obtención de ángulos y magnitudes de los mismos

Para obtener los histogramas es necesario primero obtener los ángulos y las magnitudes de los mismos, para ello será necesario calcular el valor de respuesta de los gradientes sobre cada canal de la imagen. Los autores señalan que, para cada pixel, debemos seleccionar el

canal de color que mayor valor de respuesta presente, así como el ángulo asociado. Para ello empleamos la función `get_angles_magnitudes(...)`, que, dada una lista de imágenes, obtiene la matriz de ángulos y magnitudes asociada a la misma. Cabe destacar que para obtener las derivadas los autores indican que es con el filtro de Sobel, a saber $[-1, 0, 1]$, con el que se obtienen los mejores resultados. Así pues, aplicaremos este filtro en ambos ejes de la imagen empleando la función de OpenCV `sepFilter2D(...)`. A continuación se muestra el código que implementa la función `get_angles_magnitudes(...)`:

"."

```
#Función que obtiene los ángulos y las magnitudes de los gradientes
#de cada imagen de una lista dada como argumento
def get_angles_magnitudes(img_list):
    img_angles_list = []
    img_magnitudes_list = []
    #Iteramos sobre la lista de imagenes
    for i in img_list:
        #Obtenemos el kernel de Sobel
        kernel=np.array([-1,0,1])
        #Aplicamos el kernel a la imagen para obtener las derivadas x e y
        x_gradient = cv2.sepFilter2D(np.float32(i), cv2.CV_32F,kernel,np.array([1]))
        y_gradient = cv2.sepFilter2D(np.float32(i), cv2.CV_32F,np.array([1]),kernel)
        #Aplicamos la función cartToPola para obtener las magitudes
        #y orientación de cada pixel
        magnitudes, angles = cv2.cartToPolar(x_gradient, y_gradient,\
                                    angleInDegrees=True)
        #De entre los tres valores de respuesta de un pixel, uno por cada canal,
        #obtenemos el máximo, así como el canal en el que fué encontrado
        maxs_index = np.argmax(magnitudes, axis=2)
        max_magnitudes = np.max(magnitudes, axis=2)
        #Inicializamos la lista de angulos asociados a los mayores valores
        #de respuesta
        maxs_angles = np.zeros(np.shape(maxs_index), dtype = np.float32)
        #Iteramos sobre la imagen
        for i in range(maxs_index.shape[0]):
              for j in range(maxs_index.shape[1]):
                 #Almacenamos el ángulo asociado al mayor valor de respuesta
                 maxs_angles[i][j] = angles[i][j][maxs_index[i][j]]
           #Almacenamos la matriz de ángulos y magnitudes asociadas
        img_angles_list.append(maxs_angles)
        img_magnitudes_list.append(max_magnitudes)
    return img_angles_list, img_magnitudes_list
```

3.4. Obtención de los histogramas

Una vez disponemos de los ángulos y sus magnitudes asociados a cada imagen, debemos obtener el histograma asociado a cada celda, para ello debemos discretizar el rango de valores que puede tomar un ángulo en arco $[0, 180)$, empleando para ello 9 intervalos de 20 grados que llamaremos cubetas. De esta manera, la magnitud de cada ángulo se reparte de manera ponderada entre las cubetas que lo incumben. Por ejemplo, dado un ángulo de 85^{o} y magnitud 100, el valor de la magnitud se reparte de forma que un 75 % de la magnitud

se destina a la cubeta del rango $[80, 100)$ y el $25\,\%$ restante a la cubeta de $[100, 120)$. A continuación se muestra el código que implementa esta idea de forma eficiente, esto es, empleando las propiedades de la división entera así como del módulo:

```
                                    "."
#Función que calcula el histograma de una matriz de ángulos y sus magnitudes
def get_histogram(angles_matrix, mag_matrix):

    #Inicializamos nuestro histograma de tamaño 9 a 0
    histogram = np.zeros(9, dtype = np.float32)

    #Recorremos la matriz de ángulos
    for i in range(angles_matrix.shape[0]):
        for j in range(angles_matrix.shape[1]):

            #Obtenemos los dos cubetas donde se repatirá el peso del ángulo
            bucket1 = 20 * (angles_matrix[i,j]//20)
            bucket2 = 20 * (angles_matrix[i,j]//20 + 1)

            #Calculamos la proporción de peso que recibirá la primera cubeta
            proportion = bucket2 - angles_matrix[i,j]

            #En función de la matriz de magnitudes y las propociones calculadas, calculam
            weight1 = (proportion/20)*mag_matrix[i,j]
            weight2 = ((20-proportion)/20)*mag_matrix[i,j]

            #Actualizamos las cubetas correspondientes con los pesos obtenidos
            histogram[int((bucket1/20)%9)] = histogram[int((bucket1/20)%9)] + weight1
            histogram[int((bucket2/20)%9)] = histogram[int((bucket2/20)%9)] + weight2

    return histogram
```

3.5. Obtención de las características

Para obtener el vector de características de cada imagen empleamos la función `get_features(...)`, que se encarga de construir tanto las celdas, empleando para ello la función `get_histogram(...)`, como los bloques. Para ello itera sobre cada imagen obteniendo una matriz de celdas que más tarde agrupará para obtener los bloques, finalmente devuelve una lista que contiene las matrices de vectores de características de cada imagen. A continuación se muestra el código que implementa esta idea:

```
                                    "."
```

```
#Función que obtiene las características asociadas a las imágenes de las que se
#han obtenido sus matrices de ángulos y magnitudes, dados estos como argumento, y
#atendiento a los tamaños de celda y bloque dados
def get_features(angles_list, magnitudes_list, cell_size, block_size):
   #Controlamos que los parámetros con compatibles
   assert len(angles_list) == len(magnitudes_list)
   assert np.shape(angles_list[0])[0] % (cell_size) == 0 or \
         np.shape(angles_list[0])[1] % (cell_size) == 0
   #Obtenemos la lista de ángulos en formato [0,180]
   ail = [i%180 for i in angles_list]
   aml = magnitudes_list
   #Calculamos el número de celdas por fila
   cpr = np.shape(ail[0])[0]//cell_size
   #Calculamos el número de celdas por columna
   cpc = np.shape(ail[0])[1]//cell_size
   #Inicializamos la lista de histogramas
   hist_list = []
   #Iteramos sobre la lista de matrices de angulos
   for img in range(len(ail)):
      #Inicializamos el histograma asociado a la matriz
      hist_matrix = np.zeros((cpr,cpc,9), dtype = np.float32)
      #Iteramos sobre la matriz virtual de celdas
      for i in range(cpr):
         for j in range(cpc):
            #Obtenemos la sub-matriz de angulos asociada a la celda
            angles_cell = (ail[img])[i*cell_size:(i+1)*cell_size,\
                              j*cell_size:(j+1)*cell_size]
            #Obtenemos la sub-matriz de magnitudes asociada a la celda
            mag_cell = (aml[img])[i*cell_size:(i+1)*cell_size,\
                           j*cell_size:(j+1)*cell_size]
            #Obtenemos los histogramas asociados a cada celda almacenada
            hist_matrix[i,j] = get_histogram(angles_cell, mag_cell)
      #Almacenamos la matriz de histogramas calculada
      hist_list.append(hist_matrix)
   #Calculamos el margen asociado al bloque
   bm = block_size//2
   #Inicializamos la lista de características de imágenes
   img_features_list = []
   #Iteramos sobre la lista de matrices de histogramas
   for m_hist in hist_list:
      #Inicalizamos la matriz de vectores de características
      features_vector_matrix = []
      #Iteramos sobre la matriz de histogramas actual respetando los margenes
      for i in range(bm, m_hist.shape[0]- bm):
         for j in range(bm, m_hist.shape[1]- bm):
```

Una vez obtenidas las características es necesario darles el formato que es aceptado por la función `svm.fit(...)` del módulo svm del paquete sklearn, que es la función que emplearemos para obtener una máquina de vectores de soporte entrenada en base a las

características calculadas . Para ello bastará con concatenar los vectores de características correspondientes a cada imagen, obteniendo así un único vector de tamaño 14580. A continuación se muestra el código de la función `build_SVM_features(...)`, encarga de realizar esta tarea:

```
                                    "."
#Función que transforma las características dadas como argumento al formato necesario
#para darlas como argumento a un SVM
def build_SVM_features(image_features_list):
    #Inicializamos la lista de características en formato SVM
    SVM_features = []
    #Iteramos sobre la lista de características dada como argumento
    for features in image_features_list:
        #Obtenemos la concatenación de las características contenidas en
        #las filas de la matriz de características asociada a cada imagen
        SVM_features.append(np.asarray(features).flatten())

    return np.array(SVM_features)
```

3.6. Normalización de las características

La magnitud de los gradientes varia en un amplio rango de valores dependiendo de la intensidad luminosa y el contraste con el fondo en la imagen, de esta forma, resulta efectivo normalizar el rango de valores de los descriptores de las imágenes para eliminar la escala y que los vectores de características no se vean afectados por las variaciones de luz. Considerando v como el vector de características no normalizado, $||v||_k$ como la k-norma de ese vector y ϵ un constante pequeña, a continuación se exponen tres de los modelos de normalización propuestos por los autores, cuyos resultados se comparan más adelante:

3.6.1. Normalización L2-normalization

La normalización L2 viene descrita por $v \rightarrow \frac{v}{\sqrt{||v||_k^2+\epsilon^2}}$ donde $||v||_k^2$ es la norma-2 del vector v. A continuación se muestra la función que normaliza las características según esta regla:

```
                                    "."
#Normalización L2
def L2_norm(image_features_list, epsilon):
    #Iteramos sobre la lista de características de imágenes
    for features in image_features_list:
        #Iteramos sobre las filas de la matriz de características
        for i in range(np.shape(features)[0]):
            #Aplicamos la normalización
            features[i] = features[i]/np.sqrt(np.linalg.norm(features[i])**2 + \
                                              epsilon**2)
```

3.6.2. Normalización L1-normalization

La normalización L1 viene descrita por $v \rightarrow \frac{v}{||v||_k^1+\epsilon^2}$ donde $||v||_k^1$ es la norma-1 del vector v. A continuación se muestra la función que normaliza las características según esta

regla:

```
#Normalización L1
def L1_norm(image_features_list, epsilon):
    #Iteramos sobre la lista de características de imágenes
    for features in image_features_list:
        #Iteramos sobre las filas de la matriz de características
        for i in range(np.shape(features)[0]):
            #Aplicamos la normalización
            features[i] = features[i]/(np.linalg.norm(features[i], 1) + \
                                       epsilon)
```

3.7. Normalización L1-sqrt

La normalización L1-sqrt viene descrita por $v \rightarrow \sqrt{\frac{v}{||v||_k^1+\epsilon^2}}$, es decir, la normalización L1 seguida de una raíz cuadrada. A continuación se muestra la función que normaliza las características según esta regla:

```
#Normalización L1-sqrt
def L1_sqrt(image_features_list, epsilon):
    #Iteramos sobre la lista de características de imágenes
    for features in image_features_list:
        #Iteramos sobre las filas de la matriz de características
        for i in range(np.shape(features)[0]):
            #Aplicamos la normalización
            features[i] = np.sqrt(features[i]/(np.linalg.norm(features[i], 1) + \
                                               epsilon))
```

3.8. Implementación del reconocimiento de peatones

Una vez definido el método de extración de características y entrenado el modelo de clasificación, definimos el procedimiento que permite desplazar una ventana de búsqueda por una imagen de dimensiones arbitrarias para localizar los peatones presentes en ella. Para ello emplearemos la función `recognise_pedestrian(...)`, que emplea las funciones descritas anteriormente y proporciona como salida la imagen original con los marcos asociados a los peatones encontrados. A continuación se muestra el código que implementa esta idea:

```
#Función que detecta peatones presentes en una imagen dada como argumento,
#empleando como predictor el SVM dado como argumento
def recognise_pedestrian(image, win_size, SVM):
    #Aplicamos la corrección gamma a la imagen dada como argumento
     correc_image = gamma_correction([image],0.2)[0]
     #Obtenemos una copia sobre la que realizaremos modifcaciones
     output_image = copy.deepcopy(image)
     #Inicializamos la lista que contendrá la pirámide
     #gaussiana asociada a la imagen
```

```
pyramid = [correc_image]
r_size = pyramid[0].shape[0]
c_size = pyramid[0].shape[1]
#Mientras la imagen obtenida sea de dimensión mayor que la ventana
while (r_size/1.2 > win_size[1] and c_size > win_size[0]):
   #Almacenamos la el siguiente nivel de la pirámide gaussiana
    pyramid.append(cv2.resize(cv2.GaussianBlur(pyramid[-1], (7,7), 1), \
                              (0, 0), fx = 0.75, fy = 0.75))
    #Actualizamos las variables centinela
    r_size = pyramid[-1].shape[0]
    c_size = pyramid[-1].shape[1]
#Iteramos sobre los niveles de la pirámide
for img in pyramid:
   #Iteramos sobre cada imagen
    for i in range(0, img.shape[0] - win_size[1] + 1, 4):
        for j in range(0, img.shape[1] - win_size[0] + 1, 8):
           #Extraemos la ventana a analizar
            window = img[i:i+win_size[1],j:j+win_size[0]]
            #Añadimos los bordes
            window = cv2.copyMakeBorder(window, 2, 2, 1, 1, cv2.BORDER_REPLICATE)
            #Obtenemos el kernel de Sobel
            kernel=np.array([-1,0,1])
            #Aplicamos el kernel a la imagen para obtener las derivadas x e y
            x_gradient = cv2.sepFilter2D(np.float32(window), \
                        cv2.CV_32F,kernel,np.array([1]))
            y_gradient = cv2.sepFilter2D(np.float32(window), \
                        cv2.CV_32F,np.array([1]),kernel)
            #Aplicamos la función cartToPola para obtener las
            #magitudes y orientación de cada pixel
            magnitudes, angles = cv2.cartToPolar(x_gradient, y_gradient, \
                                        angleInDegrees=True)
        #De entre los tres valores de respuesta de un pixel, uno por cada canal,
        #obtenemos el máximo, así como el canal en el que fué encontrado
            maxs_index = np.argmax(magnitudes, axis=2)
            max_magnitudes = np.max(magnitudes, axis=2)
            #Inicializamos la lista de angulos asociados a los mayores valores
        #de respuesta
            maxs_angles = np.zeros(np.shape(maxs_index), dtype = np.float32)
            #Iteramos sobre la imagen
            for a in range(maxs_index.shape[0]):
                for b in range(maxs_index.shape[1]):
                   #Almacenamos el ángulo asociado al mayor valor de respuesta
                    maxs_angles[a][b] = angles[a][b][maxs_index[a][b]]
            #Obtenemos las caracteristicas asociadas a la imagen
            features = get_features([maxs_angles], [max_magnitudes], 6, 3)
            #Normalizamos las características
            L2_norm_features = copy.deepcopy(features)
            L2_norm(L2_norm_features, 0.95)
            #Obtenemos als características en formato compatible con SVM
            L2_norm_features = build_SVM_features(L2_norm_features)
```

```
#Obtenemos la predicción del modelo
result = SVM.predict(L2_norm_features)
#Si la predicción es positiva dibujamos sobre la imagen
#el marco asociado a la ventana actual
if(result[0] == 1):
    sc = 1.33**l
    output_image_aux = cv2.rectangle(copy.deepcopy(output_image), \
          (int(j*sc),int(i*sc)), (int(j*sc+win_size[0]*sc), \
          int(i*sc + win_size[1]*sc)), (0,255,0), 2)
    draw_canvas([output_image_aux[:,:,::-1]], ["Pedestrian"], 1, 1)
```

4. Mejoras propuestas para el modelo

Los autores Navneet Dalal y Bill Triggs proponen en su publicación "Histogram of Oriented Gradients for Human Detection" un modelo en que el se lleva a cabo un entrenamiento preliminar del SVM, de forma que este primer modelo obtenido se emplea para obtener, sobre el conjunto de todas las imágenes que no contiene un peatón, todos los falsos positivos, los ejemlos difíciles, es decir, aquellas muestras de las imágenes negativas en las que el modelo preliminar da respuesta positiva. Una vez obtenidos los ejemplos difíciles, es necesario reentrenar el modelo teniéndolos en cuenta, esto es, añadiéndolos al conjunto de entrenamiento. El objetivo de este nuevo entrenamiento es que el SVM aprenda de manera mucho más profunda la clase no-persona, ya que esta es mucho mas amplia y variada que la clase persona y, mientras que un muestreo aleatorio no es suficiente y un método de fuerza bruta no puede ser llevado a cabo, el método descrito eficaz es eficaz y eficiente en cierta medida.

Sin embargo, encontramos que la mejora propuesta por los autores no es realizable en las máquinas de las que disponemos, podemos probar esto llevando a cabo una estimación del tiempo que tomaría extraer y procesar los ejemplos difíciles, así como reentrenar el modelo con el nuevo conjunto de entrenamiento aumentado. De esta manera tenemos que, dado que los autores proponen que para extraer los ejemplos difíciles cada imagen se muestree con una frecuencia de 4 en las filas y 8 en las columnas, el número medio de ventanas a analizar por imagen será (240 - 128)*(320 - 64) / 4 / 8 = 957. Comprobamos entonces de manera experimental que el tiempo empleado para analizar una imagen de tamaño medio es 2 minutos y 15 segundos, y dado que se deben analizar 1218, obtenemos un tiempo de cómputo estimado de 45.6 horas, sin considerar el tiempo que tomaría reentrenar el SVM con un conjunto de entrenamiento de tamaño mayor. Tomando esto en consideración, y calculando que reducir el tiempo de ejecución a un tiempo razonable supondría muestrear cada imagen de formar muy similar a un muestreo aleatorio, decidimos constatar la eficacia del reentrenamiento para aprender la clase no-persona extrayendo el doble de ejemplos negativos, lo que debe traducirse en un menor número de falsos positivos al analizar una imagen.

Tras la exprimentación, concluimos que aumentar el número de muestras de la clase no-persona, mejora los resultados obtenidos por el método en lo que respecta al número de falsos positivos, de esta forma, comparando los resultados sobre una misma imagen comprobamos que el modelo reentrenado obtiene un 43 % menos de falsos positivos que el modelo preeliminar. Teniendo en cuenta que los nuevos ejemplos para reentrenar el modelo han sido seleccionados de forma aleatoria y no guiada como proponen los autores, es de esperar que la mejora obtenida con la selección guiada sea mayor, sin embargo, estos resultados son suficientes para constatar la validez de la mejora propuesta. A continuación

se muestran los resultados obtenidos por el modelo preliminar y el reentrenado:

Figura 1: Imagen de consulta para ambos modelos

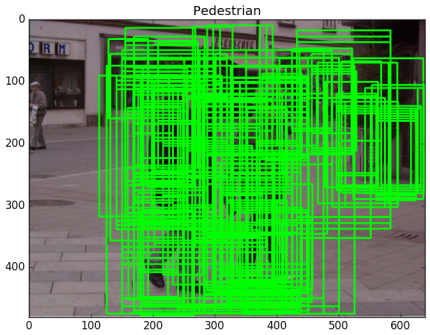


Figura 2: Totales: 314. Falsos: 232

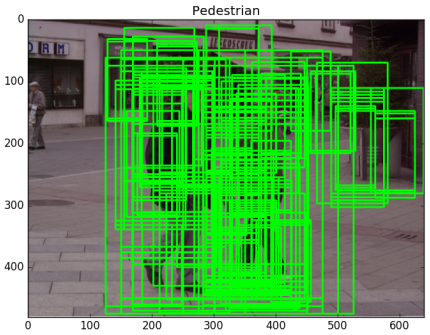


Figura 3: Totales: 185. Falsos: 134

Figura 4: Positivos encontrados por ambos métodos