



UNIVERSIDAD DE GRANADA

Visión por Computador

Proyecto final

*Laura Calle Caraballo
Javier León Palomares*

16 de enero de 2018

Índice

1. Introducción	2
2. Descripción de la técnica e implementación	2
2.1. Cálculo de gradientes	2
2.2. Obtención de histogramas	3
2.3. Normalización de histogramas	4
3. Detección de peatones	6
3.1. Obtención de datos	6
3.2. Selección del método de normalización de gradientes	7
3.3. Ajustes adicionales del modelo	8
3.4. Búsqueda de peatones en imágenes	8
3.4.1. Supresión de no máximos	10
3.4.2. Selección por votación	13
3.4.3. Incrementando el umbral	15
3.4.4. Estudio de las alternativas propuestas	17
4. Conclusiones	18

Índice de figuras

1. Original	17
2. Múltiples SVM	17
3. Umbral de 80 %	17

1. Introducción

El objetivo de este proyecto es explorar la técnica de **Histograma de Gradientes (HoG)** aplicada a la detección de peatones, descrita en *Histograms of Oriented Gradients for Human Detection* (2005) de *Dalal y Triggs*.

Para ello, hemos implementado distintos aspectos del método realizando una serie de decisiones de diseño y hemos utilizado la base de datos original para comparar resultados. Las herramientas empleadas son *Python* y las librerías *OpenCV*, *NumPy* y *scikit-learn*.

2. Descripción de la técnica e implementación

El **Histograma de Gradientes** nos permitirá extraer características relevantes de una imagen para poder proceder a su clasificación (contiene un peatón o no). Consta de varias etapas que analizaremos a continuación.

2.1. Cálculo de gradientes

El primer paso del proceso es calcular los gradientes horizontales y verticales de cada imagen. Para ello, utilizaremos un *kernel* unidimensional $[-1, 0, 1]$ centrado en cada píxel a evaluar, guiándonos por los resultados experimentales del *paper* original.

Utilizando los valores anteriores obtendremos las orientaciones de los cambios de intensidad y sus magnitudes. Posteriormente, ya que las imágenes son a color, elegiremos el canal con mayor magnitud de gradiente para cada píxel.

El código correspondiente a esta parte es:

```
def computeGradient(img, kx, ky):

    img_x = cv2.sepFilter2D(img, cv2.CV_32F, kx, ky,
                           borderType=cv2.BORDER_REPLICATE)
    img_y = cv2.sepFilter2D(img, cv2.CV_32F, ky, kx,
                           borderType=cv2.BORDER_REPLICATE)

    mag, angle = cv2.cartToPolar(img_x, img_y, angleInDegrees=True)
    angle = angle%180
    del img_x
    del img_y

    magnitudes = np.apply_along_axis(np.max, 2, mag)

    coord_0 = np.repeat(np.arange(img.shape[0]),img.shape[1])
    coord_1 = np.tile(np.arange(img.shape[1]),img.shape[0])

    indices = np.apply_along_axis(np.argmax, 2, mag)
    angles = angle[coord_0, coord_1, indices.flatten()]
    angles = angles.reshape(img.shape[:2])

    return magnitudes, angles
```

Para obtener las orientaciones de los gradientes (limitándolas al rango $[0, 180]$) y sus magnitudes hemos usado `cartToPolar`, que implementa las siguientes fórmulas:

$$\theta = \arctan \frac{g_y}{g_x}$$
$$g = \sqrt{g_x^2 + g_y^2}$$

Respecto a la selección del canal de color con mayor respuesta para cada píxel, la aproximación más directa sería utilizando dos bucles anidados para recorrer la imagen píxel por píxel, pero es posible ganar ligeramente en eficiencia aprovechando la capacidad de vectorización de NumPy: generamos todas las combinaciones de posiciones de la imagen, siendo cada una representada por `(coord_0[i], coord_1[i])`; obtenemos los índices del canal con mayor respuesta en cada píxel; finalmente, cada posición estará representada junto a su canal elegido por `(coord_0[i], coord_1[i], indices[i])`. De esta forma, podemos aprovechar la indexación por conjuntos de índices para resumir el proceso de selección en una única línea.

2.2. Obtención de histogramas

El uso de histogramas locales se basa en que la forma distintiva de un objeto se puede caracterizar muchas veces con suficiente calidad utilizando distribuciones y orientaciones de gradientes por áreas de una imagen. Además, al resumir implícitamente la información, la dimensionalidad del vector de características se reduce.

Para calcular estos histogramas, vamos a dividir la imagen en celdas cuadradas de un cierto número de píxeles (en nuestro caso, 8×8). En cuanto a los histogramas, estarán formados por 9 secciones que corresponden a intervalos de 20 grados entre 0 y 180, puesto que tomaremos las orientaciones por dirección y no por sentido. Para conocer la aportación del ángulo en un píxel, haremos una interpolación lineal que reparta su influencia entre las dos secciones más cercanas; por ejemplo, si tenemos un ángulo de 35 grados, aportará el 25% de su valor de magnitud a la sección centrada en 20 y el 75% restante a la centrada en 40.

```
def cellHistogram(cell_m, cell_o, bins = 9, max_angle = 180):
    bin_size = max_angle//bins
    histogram = np.zeros(bins)

    lower_bin = (cell_o//bin_size).astype(np.uint8)
    upper_bin = (lower_bin+1)%bins

    value_to_upper = (cell_o%bin_size) / bin_size
    value_to_lower = (1-value_to_upper) * cell_m
    value_to_upper *= cell_m

    for i in range(8):
        for j in range(8):
            histogram[lower_bin[i,j]] += value_to_lower[i,j]
            histogram[upper_bin[i,j]] += value_to_upper[i,j]

    return histogram
```

Como se puede apreciar, hacemos uso de vectorización para calcular en pocas líneas las aportaciones de todos los píxeles de la celda a las secciones del histograma.

Cabe mencionar que tanto el tamaño de las celdas como la disposición de los histogramas en 9 categorías cubriendo el rango $[0, 180]$ se han elegido así porque producen resultados de mayor calidad según el artículo. Asimismo, en el caso de las celdas un tamaño de 8×8 permite que haya un número exacto de éstas en las ventanas de 64×128 píxeles que luego usaremos para detectar peatones.

Finalmente, veamos cómo se usa la función a la hora de calcular todos los histogramas de una imagen:

```
def computeCellHistograms(border_s, magnitudes, angles, cell_s = 8,
                           bins = 9, max_angle = 180):

    end_r = magnitudes.shape[0] - border_s
    end_c = magnitudes.shape[1] - border_s

    rows = magnitudes.shape[0] - (border_s*2)
    cols = magnitudes.shape[1] - (border_s*2)

    histograms = np.zeros((rows//cell_s, cols//cell_s, bins), np.float)

    for i in range(border_s, end_r, cell_s):
        for j in range(border_s, end_c, cell_s):
            # pixel coordinates to cell coordinates
            histograms[(i-border_s)//cell_s, (j-border_s)//cell_s] = \
                cellHistogram(magnitudes[i:i+cell_s, j:j+cell_s],
                              angles[i:i+cell_s, j:j+cell_s], bins, max_angle)

    return histograms
```

Lo primero que hacemos es eliminar el borde que pueda tener la imagen hasta que sus dimensiones sean múltiplos del tamaño de celda. Esto se hace principalmente porque los ejemplos positivos de peatones en los conjuntos de entrenamiento y test tienen un tamaño ligeramente mayor al de 64×128 que vamos a necesitar.

Posteriormente, recorremos la imagen en saltos de 8 píxeles de izquierda a derecha y de arriba abajo para ir obteniendo los histogramas de toda la malla de celdas.

2.3. Normalización de histogramas

La alta variabilidad de contextos y grados de iluminación que existe en la realidad provoca que los gradientes que calculamos se muevan en rangos muy amplios. Puesto que esto puede añadir una complejidad significativa a la ya difícil tarea de distinguir entre una clase y todo lo demás, es necesario el uso de normalización para tener una escala unificada de magnitudes entre 0 y 1.

En nuestro caso, esta normalización se realiza agrupando las celdas en bloques cuadrados (por defecto, de 2×2 celdas) y aplicando el proceso con una cierta superposición entre dichos bloques (por defecto, 50%). La superposición, aunque pueda introducir redundancia, aporta calidad al proceso y mejora los resultados, guiándonos por las conclusiones de los autores de la técnica.

Según el cálculo realizado para normalizar, podemos distinguir entre las tres variantes probadas:

- División entre la norma L1:

```
def norm_1(x):  
    norm = np.linalg.norm(x,0)  
    return x/norm if norm != 0 else x
```

- División entre la norma L2:

```
def norm_2(x):  
    norm = np.linalg.norm(x)  
    return x/norm if norm != 0 else x
```

- División preliminar entre la norma L2, recorte de valores superiores a 0.2 y división entre su nueva norma L2 (llamada norma L2-Hys en el artículo):

```
def norm_2_hys(x):  
    x = norm_2(x)  
    x[x > 0.2] = 0.2  
    return norm_2(x)
```

La forma de aplicar lo anterior se traduce en la siguiente función:

```
def normalizeHistograms(histograms, norm_f = norm_2, block_size = 2,  
                        overlapping = 0.5):  
    step = int(block_size*(1-overlapping))  
    normalized = []  
    for i in range(0,histograms.shape[0]-block_size+1, step):  
        for j in range(0,histograms.shape[1]-block_size+1, step):  
            normalized.extend((norm_f(histograms[i:i+block_size,  
                                         j:j+block_size].flatten())))  
    return np.asarray(normalized)
```

La función recibe como parámetro el conjunto de histogramas, así como la normalización que aplicará, el tamaño de los bloques y el solapamiento entre los mismos. Tras calcular el intervalo entre cada par de bloques, los recorremos y añadimos cada conjunto resultante de histogramas normalizados al vector de características de la imagen.

3. Detección de peatones

3.1. Obtención de datos

Para la detección se ha utilizado un tamaño de ventana de 64×128 píxeles, cuya forma permite abarcar personas aproximadamente erguidas. Asimismo, dentro de las ventanas con ejemplos positivos de entrenamiento y test, parte de este tamaño es un borde alrededor del peatón que añade contexto útil; los autores comprobaron que reducir este borde causaba una pérdida notable de precisión.

Con los ejemplos positivos ya proporcionados, nuestra próxima tarea será conseguir ejemplos negativos del mismo tamaño a partir de las fotografías del conjunto de datos que no contienen personas. Siguiendo el procedimiento original, vamos a extraer 10 ventanas aleatorias de cada una de esas imágenes. Para no tener que generar miles cada vez que queramos entrenar un clasificador, las guardamos una única vez:

```
def extractNegativeWindows(path, dst_path):
    for name in os.listdir(path):
        img = cv2.imread(os.path.join(path, name))
        for i in range(10):
            f = random.randint(0, img.shape[0]-128)
            c = random.randint(0, img.shape[1]-64)
            cv2.imwrite(dst_path+"/"+str(i)+"_"+name, img[f:f+128, c:c+64])
```

Una vez tenemos todas las imágenes necesarias, es el momento de aplicar el proceso descrito en el apartado anterior a cada una de ellas para obtener las características con las que trabajarán los clasificadores de peatones. En código esto se traduce en recorrer los directorios de ejemplos positivos y negativos y utilizar las distintas funciones que calculan un **Histograma de Gradientes**:

```
def obtainDataFeatures(deriv_kernel, pos_path, neg_path, norm_f = norm_2):
    kx = deriv_kernel[0]
    ky = deriv_kernel[1]

    pos_dir = os.listdir(pos_path)
    neg_dir = os.listdir(neg_path)

    descriptor_length = (15*7)*(4*9)
    features = np.zeros((len(pos_dir) + len(neg_dir), descriptor_length))
    labels = np.repeat(np.asarray([1,0]), [len(pos_dir), len(neg_dir)])
    i = 0

    for name in pos_dir:
        img = cv2.imread(os.path.join(pos_path, name))

        magnitudes, angles = computeGradient(img, kx, ky)

        border_s = (img.shape[0]-128) // 2
        histograms = computeCellHistograms(border_s, magnitudes, angles, 8)
```

```
descriptor = normalizeHistograms(histograms,norm_f=norm_f)
features[i,:] = descriptor
i+=1

for name in neg_dir:
    img = cv2.imread(os.path.join(neg_path, name))

    magnitudes, angles = computeGradient(img, kx, ky)

    border_s = (img.shape[0]-128) // 2
    histograms = computeCellHistograms(border_s, magnitudes, angles, 8)
    descriptor = normalizeHistograms(histograms,norm_f=norm_f)
    features[i,:] = descriptor
    i+=1

return features, labels
```

Por eficiencia, creamos la matriz que contendrá los vectores de características de todos los ejemplos de entrenamiento en lugar de extenderla a cada paso. Esto requiere conocer el tamaño de dichos vectores de antemano, algo que nos podemos permitir si analizamos cómo hacemos el cálculo del **Histograma de Gradientes**: utilizamos ventanas de 64×128 píxeles, celdas de 8×8 píxeles y una superposición del 50% en bloques de 2×2 celdas; si en una ventana caben 16 filas de 8 celdas, esto significa que hay 15 filas de 7 bloques cada una; además, cada bloque contiene 4 celdas, que producen histogramas de 9 casillas. Esto produce un total de $15 \cdot 7 \cdot 4 \cdot 9$ características.

Tal como se mencionó anteriormente, las imágenes positivas ya incluidas en el conjunto de datos tienen un borde que incrementa un poco el tamaño, por lo que antes de procesar cada una hay que conocer cuánto vale ese borde para que `computeCellHistograms` trabaje sobre 64×128 .

Llamaremos a esta función dos veces: una para crear los datos de entrenamiento y otra para crear los datos de test. En total, el conjunto de entrenamiento consta de 2416 ejemplos positivos y 12180 negativos; el conjunto de test tiene 1132 muestras positivas y 4530 negativas.

3.2. Selección del método de normalización de gradientes

Como hemos dicho, podemos optar entre tres normalizaciones distintas, por lo que el primer paso es elegir una de ellas para enfocar futuras mejoras. Con este objetivo en mente, vamos a comparar la calidad de las predicciones de test usando un *SVM* preliminar con $C = 1$ (valor por defecto):

Normalización	Precisión
Norma L1	96.556 %
Norma L2	98.446 %
Norma L2-Hys	98.093 %

Tabla 1: Precisión (aciertos frente al total de predicciones) con los distintos métodos de normalización.

A la vista de los resultados, elegimos la norma L2 como método de normalización a partir

de ahora. En cuanto a los porcentajes de acierto, son aceptables, pero si tenemos en cuenta que en cada imagen real habremos de analizar un gran número de ventanas para encontrar personas, nos interesa tratar de mejorar estas cifras.

3.3. Ajustes adicionales del modelo

En esta sección vamos a buscar brevemente alguna forma de incrementar la precisión del modelo, para lo que contemplaremos dos factores: el parámetro C (que controla la penalización de los errores de clasificación) y el hecho de que las clases no están equilibradas (la clase no-peatón tiene muchos más ejemplos porque su variabilidad es mayor). Para el parámetro C vamos a probar el valor de 0.01 que usan los autores, y para el equilibrio de clases vamos a ponderar su peso de forma inversa a su frecuencia.

Variante	Precisión
Sin modificaciones	98.446 %
Con $C = 0.01$	98.658 %
Con ponderación de clases	98.410 %
Con ambas modificaciones	98.340 %

Parece que cambiando el valor del parámetro C hemos conseguido cierta mejora. El cambio desde 1 hasta 0.01 significa que penalizamos menos los errores de clasificación a cambio de encontrar un margen mayor entre las dos clases y el hiperplano que trata de separarlas, por lo que quizás hemos obtenido esta calidad a cambio de sacrificar algunos aciertos puntuales.

3.4. Búsqueda de peatones en imágenes

Una vez hemos intentado aumentar la capacidad de predicción base del modelo, vamos a probarlo en un entorno más realista: encontrar peatones en una imagen arbitraria. El proceso consistirá en recorrer la imagen en varias escalas con una ventana deslizante a intervalos regulares; de esta forma, aumenta el tiempo de cómputo pero también la probabilidad de encontrar a una persona que esté presente.

Una primera aproximación es la siguiente:

```
def scanForPedestriansSimple(img, classifier, deriv_kernel, norm_f, stride = 8):  
  
    dims = (int(img.shape[1]//1.2), int(img.shape[0]//1.2))  
    pyramid = [img]  
    windows = []  
  
    while dims[0] > 128 and dims[1] > 64:  
        blurred = cv2.GaussianBlur(pyramid[-1], ksize=(5,5), sigmaX = 0.6)  
        pyramid.append(cv2.resize(src=blurred, dsize=dims))  
        dims = (int(dims[0]//1.2), int(dims[1]//1.2))  
  
    scale = 1  
  
    for level in pyramid:  
        # Margin in each side: top, bottom, left, right  
        mt = int((level.shape[0] % 8) // 2)
```

```

mb = int((level.shape[0] % 8) - mt)
ml = int((level.shape[1] % 8) // 2)
mr = int((level.shape[1] % 8) - ml)

magnitudes, angles = computeGradient(level[mt:level.shape[0]-mb,
                                         ml:level.shape[1]-mr],*deriv_kernel)
histograms = computeCellHistograms(0,magnitudes,angles)

for i in range(0,histograms.shape[0]-15):
    for j in range(0,histograms.shape[1]-7):
        window_features = normalizeHistograms(histograms[i:i+16,j:j+8],
                                              norm_f=norm_f)

        if classifier.predict([window_features])[0]:
            # coordinates x,y
            windows.append(np.array([j*8*scale,i*8*scale,
                                     (j*8+64)*scale, (i*8+128)*scale, 1]))

scale *= 1.2

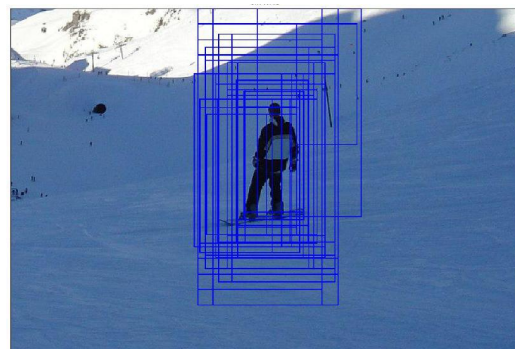
return windows

```

En primer lugar, hay que crear las diferentes escalas para las que vamos a pasear la ventana deslizante; el artículo original sugiere dividir entre 1.2 en tantos pasos sucesivos como se pueda antes de tener una imagen de menores dimensiones que la ventana.

A continuación, nos centraremos en cada nivel de la pirámide haciendo lo siguiente: tras ajustar el tamaño para que no sobren píxeles al pasear la ventana, calculamos los histogramas de toda la imagen; para cada posición posible, normalizamos los histogramas de sus celdas (obteniendo así sus características) y preguntamos al clasificador; si responde positivamente, será guardada en su localización y proporción respecto a la imagen original.

Algunos resultados de la ejecución se ven a continuación:





Se observa de forma clara que la tasa de error que obteníamos se convierte en un problema cuando evaluamos miles de ventanas por imagen. Por ello, es preciso tratar de dar soluciones.

3.4.1. Supresión de no máximos

Aunque no arreglará el problema de la calidad del modelo, una alternativa para descartar muchas de las ventanas redundantes es la supresión de no máximos ¹ (NMS). La idea aquí es aprovechar el grado de confianza del clasificador en que una ventana contenga un peatón y descartar todas aquéllas que se solapen en cierta medida con una de mayor confianza. Veamos cómo hacer esto:

```
def non_maximum_suppression(windows, overlap_threshold):
    if not len(windows):
        return np.array([])
    # windows[:,0], windows[:,1] contain x,y of top left corner
    # windows[:,2], windows[:,3] contain x,y of bottom right corner
    I = np.argsort(windows[:,4])[:,::-1]
    area = (windows[:,2]-windows[:,0]+1) * (windows[:,3]-windows[:,1]+1)
    chosen = []

    while len(I):
        i = I[0]
        # Dims of intersections between window i and the rest
        width = np.maximum(0.0, np.minimum(windows[i,2], windows[I,2])-
            np.maximum(windows[i,0], windows[I,0])+1)
        height = np.maximum(0.0, np.minimum(windows[i,3], windows[I,3])-
            np.maximum(windows[i,1], windows[I,1])+1)
        overlap = (width*height).astype(np.float32)/area[I]
        mask = overlap<overlap_threshold
        I = I[mask]
        if mask.shape[0]-np.sum(mask) > 1 :
            chosen.append(i)
    return windows[chosen]
```

¹Según la implementación de *Tomasz Malisiewicz* encontrada en su [blog personal](#)

Inicialmente aprovechamos los valores de confianza de las ventanas para ordenarlas de mayor a menor (nótese que *I* es un vector de índices) y calculamos también sus áreas.

Ahora comienza el proceso de descarte. En cada iteración seleccionamos la ventana con mayor confianza de entre las que quedan sin procesar y calculamos la proporción de su solapamiento con las demás (dividiendo entre sus áreas). En caso de ocurrir alguna superposición, se guarda la ventana en cuestión y se eliminan todas aquellas que se superpongan más de un cierto umbral (fijado aquí a 0.3 mediante experimentación). El motivo de no guardar la ventana si ninguna otra se solapa con ella (no incluido en la idea de base) es que en el entorno de un peatón suele haber varias ventanas redundantes que lo detectan, por lo que de esta forma podemos descartar algunos falsos positivos.

Como consecuencia de añadir la supresión de no máximos, el procedimiento de búsqueda se ve ligeramente alterado:

```

for level in pyramid:
    # Margin in each side: top, bottom, left, right
    mt = int((level.shape[0] % 8) // 2)
    mb = int((level.shape[0] % 8) - mt)
    ml = int((level.shape[1] % 8) // 2)
    mr = int((level.shape[1] % 8) - ml)

    magnitudes, angles = computeGradient(level[mt:level.shape[0]-mb,
                                           ml:level.shape[1]-mr],*deriv_kernel)
    histograms = computeCellHistograms(0,magnitudes,angles)

    for i in range(0,histograms.shape[0]-15):
        for j in range(0,histograms.shape[1]-7):
            window_features = normalizeHistograms(histograms[i:i+16,j:j+8],
                                                  norm_f=norm_f)
            confidence = classifier.decision_function([window_features])[0]
            if confidence > 0:
                # coordinates x,y
                windows.append(np.array([j*8*scale,i*8*scale,
                                         (j*8+64)*scale, (i*8+128)*scale, confidence]))

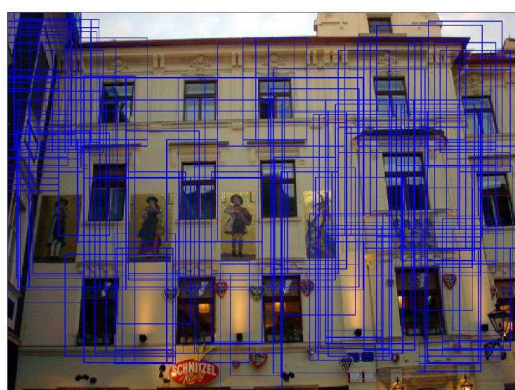
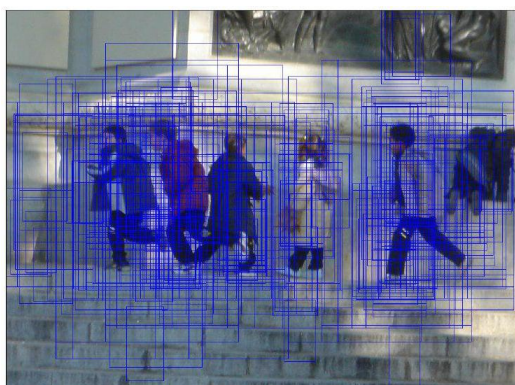
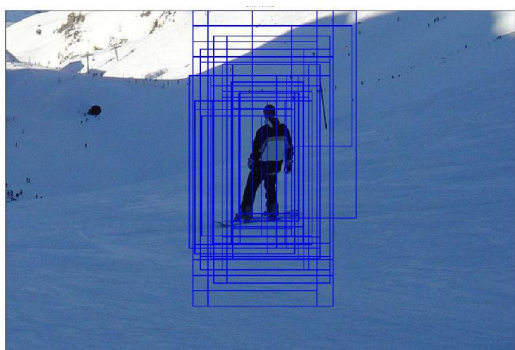
    scale *= 1.2

return non_maximum_suppression(np.asarray(windows), 0.3)

```

En la pregunta al clasificador hemos usado `decision_function`, que devuelve la distancia al hiperplano separador, porque necesitamos medidas de confianza en las predicciones para utilizarlas en `non_maximum_suppression`. Si `decision_function` da un valor mayor que 0, la muestra está en la clase positiva; si da menor que 0, está en la clase negativa. Cabe mencionar que, cuanto más lejos esté un valor del hiperplano, más seguros podemos estar de que pertenece a la clase correspondiente.

Como prueba del efecto que tiene la supresión de no máximos, comparemos los resultados anteriores (izquierda) con los obtenidos usando este filtro (derecha):



De lo anterior podemos concluir que la supresión de no máximos es de gran ayuda, pero que la utilidad del proceso que realiza depende del modelo. En nuestro caso, aún queda un número notable de imprecisiones.

Para concluir el análisis de esta mejora, veamos la penalización en tiempo de ejecución, tanto considerando el proceso tal cual como añadiendo el posterior dibujo:

Imagen	Tiempo original (s)	Tiempo NMS (s)	Tiempo original + dibujado (s)	Tiempo NMS + dibujado (s)
Imagen 1	65.83891	66.18080	70.66711	70.39946
Imagen 2	12.39770	12.29394	14.35838	13.94862
Imagen 3	68.98459	70.52982	74.14415	73.80650
Imagen 4	20.54199	20.53334	23.24827	22.88052

En la tabla se aprecia que la pérdida de velocidad es mínima, y que incluso obtenemos beneficio si abarcamos también la posterior fase de dibujo de las ventanas.

3.4.2. Selección por votación

Asumiendo el uso de *NMS*, otra manera de intentar arreglar el problema es utilizar varios clasificadores y elegir la clase de una imagen por mayoría simple. Veamos cómo se han entrenado:

```
def train_classifier_set(features_yes, features_no, n):
    classifiers = []
    c = svm.LinearSVC(C = 0.01)
    labels = np.repeat(np.asarray([1,0]), [len(features_yes),len(features_no)])

    for i in range(n):
        samples_no = random.sample(range(len(features_no)), len(features_yes))
        c.fit(np.concatenate((features_yes, features_no[samples_no])), labels)
        classifiers.append(c)
    return classifiers
```

Recibiendo como parámetro los conjuntos completos de muestras, entrenamos cada uno de los n clasificadores con todos los ejemplos positivos y con un subconjunto aleatorio de ejemplos negativos, de forma que las clases estén equilibradas.

Para la decisión por mayoría simple, lo único que hemos de hacer es sumar los votos y dividir por el número de clasificadores; si supera 0.5, se acepta como positivo:

```
def heavy_classifier(window, classifier_list):
    votes = 0
    for classifier in classifier_list:
        votes += classifier.predict(window)
    return votes/len(classifier_list)
```

Finalmente, actualizamos la parte del código en la que buscamos ocurrencias de peatones:

```
for i in range(0,histograms.shape[0]-15):
    for j in range(0,histograms.shape[1]-7):
        window_features = normalizeHistograms(histograms[i:i+16,j:j+8],
                                                norm_f=norm_f)
        confidence = heavy_classifier([window_features], classifiers)
        if confidence > 0.5:
            # coordinates x,y
            windows.append(np.array([j*8*scale,i*8*scale,
                                    (j*8+64)*scale, (i*8+128)*scale, confidence]))
```

Como vemos, el único cambio es que la clasificación ahora toma en cuenta el voto por mayoría, siendo afirmativo en caso de que su proporción supere el 50%.

Comparemos ahora los nuevos resultados (izquierda) con los obtenidos en la supresión de no máximos (derecha):





El sistema de votos, probado con 11 *SVMs*, parece dar peores resultados en cuanto a falsos positivos. Sin embargo, al final de la sección analizaremos todas las alternativas para sacar algunas conclusiones más.

3.4.3. Incrementando el umbral

Otra posibilidad para rebajar la tasa de falsos positivos es elevar el umbral a partir del cual consideramos que una imagen contiene un peatón. La desventaja de esto es que aumentará el número de falsos negativos, pero el equilibrio entre ambos tipos de errores es una cuestión que trataremos más adelante.

La librería *scikit-learn* contiene varias formas de utilizar un modelo *SVM*. La implementación más eficiente y escalable no ofrece la posibilidad de devolver la confianza (en tanto por 1) de la predicción, por lo que sólo podríamos usar la implementación más lenta, que sí tiene tal funcionalidad.

Sin embargo, realizando pruebas obtenemos que una regresión logística tiene un acierto del 98.6% en test, muy similar a nuestro mejor resultado. Además, permite obtener la asignación de probabilidades de cada clase, por lo que lo usaremos como filtro con un umbral superior a 0.5 (el estándar).

Para hacer esto, sólo tenemos que volver a cambiar la misma sección de código que antes: añadiremos un condicional que sólo evaluará la ventana con el *SVM* si pasa el filtro del 80% de confianza de la regresión logística. Si el *SVM* también responde afirmativamente, guardaremos la ventana.

```

for i in range(0,histograms.shape[0]-15):
    for j in range(0,histograms.shape[1]-7):
        window_features = normalizeHistograms(histograms[i:i+16,j:j+8],
                                                norm_f=norm_f)
        prob = simple_classifier.predict_proba([window_features])

        if prob[0,1] > 0.8:
            conf = classifier.decision_function([window_features])[0]
            if conf > 0:
                # coordinates x,y
                windows.append(np.array([j*8*scale,i*8*scale,
                                         (j*8+64)*scale, (i*8+128)*scale, conf]))

```

Comprobemos los cambios que produce esta modificación (izquierda) frente a la supresión de no máximos simple (derecha).





A raíz de lo que observamos, podríamos pensar que obtenemos algún tipo de mejora. No obstante, con sólo 4 imágenes no es posible afirmar con rotundidad que sea una tendencia, ni aquí ni en los cambios anteriores. Por lo tanto, en el siguiente apartado echaremos un vistazo a las matrices de confusión, que contienen información más generalizable acerca del rendimiento.

3.4.4. Estudio de las alternativas propuestas

Dejando a un lado la supresión de no máximos, cuya utilidad es evidente pero no influye en la calidad de la clasificación, vamos a ver las matrices de confusión del modelo original, de la alternativa por votos en mayoría simple y de la opción de incremento de umbral:

		Valores reales	
		N	P
Predicciones	N	4510	56
	P	20	1076

Figura 1: Original

		Valores reales	
		N	P
Predicciones	N	4449	22
	P	81	1110

Figura 2: Múltiples SVM

		Valores reales	
		N	P
Predicciones	N	4522	87
	P	8	1045

Figura 3: Umbral de 80 %

Como se puede observar, la decisión mediante *SVM* simple proporciona un menor número de errores en general; si le damos la misma importancia a los dos tipos de fallos en la clasificación, éste es el modelo a usar. Sin embargo, si decidiésemos penalizar más un falso negativo, la votación de múltiples *SVM* es más adecuada; de la misma forma, si quisiéramos evitar los falsos positivos, elevar el umbral como lo hemos hecho mediante una regresión logística sería nuestra elección.

En nuestro caso, ya que no tenemos información específica acerca del contexto donde se aplicaría el modelo, vamos a elegir el que proporciona una mayor tasa de acierto, el *SVM* simple.

4. Conclusiones

La técnica de **Histograma de Gradientes**, aunque ya existía anteriormente, supuso un avance notable en la detección de personas a partir de la publicación del artículo en el que se basa este trabajo. Hemos podido comprobar que alcanza una alta tasa de acierto, si bien aún tiene limitaciones en su aplicación práctica.

Algunas de estas limitaciones vienen dadas por la implementación, realizada en un lenguaje de alto nivel y de propósito general como Python. En nuestras pruebas se puede observar un tiempo de ejecución notablemente alto, nada adecuado para aplicaciones con requisitos de tiempo real.

Asimismo, una buena selección de zonas prometedoras aceleraría el proceso, evitando una búsqueda exhaustiva que escala de forma cuadrática con el tamaño de la imagen.

Por otra parte, y como se está comprobando en los últimos años, parece existir un límite en la calidad de las características que puede extraer un ser humano, viéndose mejores resultados cuando se emplean redes neuronales artificiales para este fin.

Sin embargo, una vez enumeradas algunas razones por las que se podría complicar su aplicación real, es destacable la alta tasa de acierto lograda con clasificadores sencillos (más del 98%); incluso, con un poco de elaboración posterior a la clasificación, podemos ver resultados aproximados. Parece ser, pues, que una distribución de gradientes es lo suficientemente informativa a la vez que compacta como para permitir ajustes de cierta calidad con poco esfuerzo.