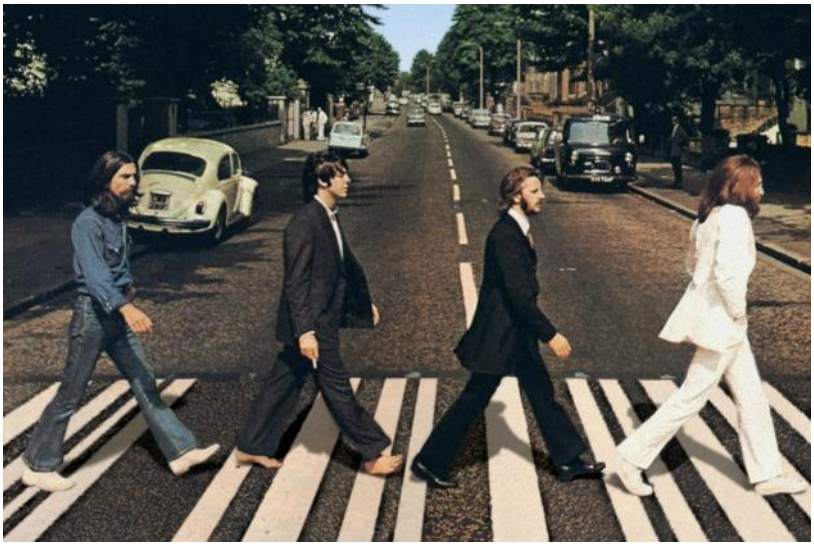


IMPLEMENTACIÓN DE HOG PARA DETECCIÓN DE PEATONES

VISIÓN POR COMPUTADOR

ABSTRACT

Implementación del descriptor HOG de Navneet Dalal and Bill Triggs para la detección de peatones en imágenes

Miguel Ángel López Robles y Rafael Sanjuan Aguilera

Trabajo Final

Introducción

Se quiere implementar un sistema de detección de peatones en imágenes. De tal forma que el software implementado podrá responder si encuentra un peatón en una imagen entrada y delimitar aproximadamente el lugar en el que se encuentra con una ventana rectangular.

Para lograr esta detección de peatones se va a entrenar una SVM, que a partir del descriptor Histogramas de Gradientes Orientados (HOG) detallado en el trabajo de Dalal y Triggs (Dalal & Triggs, n.d.), tome la decisión de la existencia de un peatón. Los pasos seguidos para lograr este objetivo son los explicados en la documentación y que vamos a exponer a continuación.

Implementación del descriptor

Podemos encontrar tanto los detalles de la implementación del descriptor, tanto como los efectos de las decisiones tomadas afectan a la eficacia de los resultados en el paper (Dalal & Triggs, n.d.). El descriptor se construye con los siguientes pasos:

Corrección gamma

Aplicar corrección gamma puede afectar ligeramente a los resultados, pero no es uno de los factores más relevantes, la formula usada ha sido la siguiente:

$$I_o = I_i^{\frac{1}{\gamma}}$$

Donde $\boldsymbol{I_o}$ es la imagen resultado, $\boldsymbol{I_i}$ es la imagen de entrada y $\boldsymbol{\gamma}$ es el valor de gamma. Valores > 1 aumentaran la intensidad de los pixeles de la imagen. Esta corrección puede ayudarnos a resaltar zonas oscuras, donde el peatón no se reconoce con claridad. El objetivo de esta corrección va a ser intentar corregir las distintas iluminaciones que encontramos en las imágenes.

Computo de los Gradientes

Como el propio nombre indica, Histogramas de Gradientes Orientados, se utiliza la información de los gradientes como información para crear el descriptor. Los autores indican en la documentación que han experimentado con distintos kernels y escalas de filtrado gaussiano. La más eficaz al final resulta ser también la más simple, calcular los gradientes con la máscara de derivadas centrada [-1,0,1] y $\sigma = 0$ da los mejores resultados. Esto nos aporta un grado de simplicidad. Por lo que simplemente aplicaremos dicho filtro en ambos ejes, X e Y

```
1.  # Compute centered horizontal and vertical gradients with no smoothing
2.      kernel = np.asarray([-1, 0, 1])
3.      kernel_vacio = np.asarray([1])
4.
```

```
    gradientsx = cv2.sepFilter2D(img, -1, kernel,kernel_vacio)
    #showImgAndWait("Gradientes", gradientsx)

    gradientsy = cv2.sepFilter2D(img, -1,kernel_vacio, kernel)
    #showImgAndWait("Gradientes", gradientsy)
```

Se aplica la máscara de forma separada en ambos ejes para poder obtener la magnitud y el ángulo de los gradientes, que obtenemos con la ayuda de la función que proporciona OpenCV:

```
# Calculate the magnitudes and angles of the gradients
    magnitude, angle = cv2.cartToPolar(gradientsx, gradientsy, angleInDegrees=True)
```

Ahora ya disponemos de dos matrices, con un tamaño igual a la imagen resultante, donde tenemos por un lado los ángulos y por el otro las magnitudes. Los autores comentan que si se trabaja con imágenes a color debemos de separar los tres canales y pasar a un único canal. Para ello deberemos quedarnos con el canal que nos indique un gradiente con mayor magnitud. Por tanto, deberemos consultar en cada pixel las magnitudes y quedarnos con un solo canal, el cual sea el máximo. Para ello usamos el siguiente proceso:

```
intensity = np.argmax(magnitude, axis=2)

x, y = np.ogrid[:intensity.shape[0], :intensity.shape[1]]
max_angle = angle[x, y, intensity]
max_magnitude = magnitude[x, y, intensity]
```

Discretización de los ángulos e histogramas

El siguiente paso es la “no-linealidad fundamental del descriptor”, cada pixel aporta un voto ponderado para formar un histograma de la magnitud de los gradientes en cada orientación. Los pixeles se agrupan en regiones de vecinos llamadas ***celdas*** sobre las que formamos el histograma. En esta parte influyen varias decisiones de diseño.

En primer lugar, debemos decidir el tamaño de las celdas, es decir cuántos pixeles debemos de coger para generar el histograma. Los autores nos indican que para el problema que tratamos el mejor valor para este cometido es realizar celdas de tamaño 6x6 pixeles.

Con esta decisión, la siguiente a tomar es como discretizar los ángulos. Los autores se refieren a esto como **bin**, y comentan que tenemos varias opciones desde 3-9 bin si usamos ángulos sin signo o de 6-18 para ángulos con signo. Los ángulos sin signo es usar solo la semicircunferencia superior del círculo de grados, es decir ángulos de 0-180. Un bin de 9, con ángulos de 0-180, se propone como la solución que ofrece un mejor desempeño. Para la transformación se realiza la siguiente operación

$$angulo_{0-180} = (\,360 - angulo_{0-360})\,\%\,180$$

A continuación, bastará con coger los pixeles implicados en cada celda y calcular el histograma. Para realizar la discretización en bin, de los ángulos, debemos realizar una proporción del ángulo que queremos que vote. Como tenemos 9 bin es muy probable encontrar un ángulo entre varios. Por lo que debemos calcular la distancia del ángulo a los dos bin más cercanos y realizará el voto en proporción a la distancia. Por ejemplo, un ángulo de 30º estaría entre los bins de 20º y 40º. Por tanto, votará con su magnitud en ambos con una proporción de 0,5. De este modo se busca evitar el aliasing.

Por último, debemos de tratar como se realiza el voto de cada pixel, ¿usamos la magnitud, su raíz cuadrada, el cuadrado? Esto también es tratado por los autores, los cuales comentan que la solución más sencilla y con mejor rendimiento es usar la magnitud tal cual. Con todos estos factores se construye la siguiente función que nos permite calcular el histograma relativo a una celda:

```
#function to calculate histograms
def genHistogram(angles, magnitudes):
    histogram = np.zeros(9, dtype=np.float32)

    for i in range(angles.shape[0]):
        for j in range(angles.shape[1]):
            # Find the two nearests bins.
            bin1 = int( angles[i, j] // 20 )
            bin2 = int(( (angles[i, j] // 20) + 1) % 9)

            # calculate the proportional vote for the two affected angles
            prop = (angles[i, j] - (bin1 * 20)) / 20

            # calculate the value vote for the two affected bin
            vote1 = (1 - prop) * magnitudes[i, j]
            vote2 =  prop * magnitudes[i, j]

            histogram[bin1] += vote1
            histogram[bin2] += vote2

    return histogram
```

Bloques y normalización

Como la intensidad de los gradientes varia bastante debido a variaciones locales de iluminación, la normalización local resulta fundamental para obtener buenos resultados. Se propone entonces dividir la imagen agrupando las celdas en ***bloques*** de celdas vecinas.

Estos bloques se superponen de forma densa para que cada celda contribuya a varios componentes del descriptor final, siendo la contribución de la celda también ponderada en cada una de las contribuciones según la distancia al centro del bloque. Cuando tengamos todos los bloques normalizados, cada bloque estará formado por un número de histogramas, y con el paso a un vector de todos los bloques obtenemos el descriptor final de la imagen. Vamos a ir analizando poco a poco cada paso para entender mejor el proceso que se realiza.

Como en el caso de las celdas, también se debe decidir el tamaño del bloque, es decir cuántas celdas agrupamos. Usamos el valor óptimo indicado por los autores, 3x3 celdas.

En primer lugar, debemos tener en cuenta que ahora trabajamos con celdas, es decir con histogramas. Los autores indican que realizar un filtro Gaussiano para dar más peso a los histogramas centrales y menos a los que se encuentran en los bordes, puede incrementar el rendimiento del método. Para ello hemos aplicado una máscara a cada bloque con una sigma igual a 0.5*tamaño del bloque.

A continuación, se plantean 4 tipos de normalizaciones a nivel local de cada bloque. Tenemos 4 opciones: L1, L1 sqrt, L2, L2 hys. Los detalles de cada normalización podemos encontrarlos en la documentación o en la propia implementación. Los autores comentan que el método que peores resultados ofrece es la normalización L1, pero en nuestra experimentación hemos observado que las 4 son similares y la diferencia no es tan significativa. Finalmente usaremos la L2 como se indica en el paper.

Por último, debemos comentar el tema del solapamiento. Se expone como una forma de introducir redundancia y hacer más robusto el método. Es decir, en nuestro caso, si pensamos en una dimensión, cogemos 3 celdas para un bloque y para el siguiente bloque cogeremos los 3 siguientes. Con el solapamiento se cogerían 3 celdas para el primer bloque y saltaríamos una única celda para construir el siguiente, es decir se repiten 2 celdas. En el paper se habla de un solapamiento de 3/4, el cual ofrece una mejor performance. En nuestro caso para evitar tener que recalcular celdas vamos a usar un solapamiento 2/3.

Con todos estos datos pasamos a tener la siguiente función para generar los bloques. Los cuales, colocados como vector, nos proporcionan el descriptor de la imagen.

```
1.  #function to generate the block, with Gaussian mask and normalization
2.  # we use the overlap 2/3
3.  def genBlocks(block_size, cells, stride=1, sigma=0.5, normalization_tipe=2, threshold=0
    ):
4.      ini = block_size // 2
5.      fin = block_size // 2
6.
7.      if block_size % 2 != 0:
8.          fin = fin + 1
9.
10.     sigma = block_size *0.5
11.     first_stop = ini
12.     second_stop = ini
13.
14.     if np.shape(cells)[0] % block_size == 0:
15.         first_stop = first_stop - 1
16.
17.     if np.shape(cells)[1] % block_size == 0:
18.         second_stop = second_stop - 1
19.
20.     #calculate the Gaussian mask
```

```
    #gaussian = cv2.getGaussianKernel(block_size, sigma)
    #gauss2d = gaussian*np.transpose(gaussian)
    blocks = []
    for i in range( 0, np.shape(cells)[0] - block_size, stride):
        for j in range( 0, np.shape(cells)[1] - block_size, stride):

            block = np.array(cells[i :i + block_size, j:j + block_size])

            #apply mask Gaussian
            #for x in range(0,block.shape[0]):
             #   for y in range(0,block.shape[1]):
              #      for h in range(0,9):
               #         block[x, y, h] = block[x, y, h] * gauss2d[x, y]

            block = block.flatten()
            #normalization
            if normalization_tipe == 0:
                block =  normalize_L1(block, threshold)
            elif normalization_tipe == 1:
                block = normalize_L1_sqrt(block, threshold)
            elif normalization_tipe == 2:
                block = normalize_L2(block, threshold)
            elif normalization_tipe == 3:
                block = normalize_L2_Hys(block, threshold)


            blocks.append(block)

    blocks = np.array(blocks, dtype=np.float32)
    return blocks
```

Función final

Con todos los pasos descritos anteriormente, tenemos las herramientas necesarias que nos permiten poder generar el descriptor de una imagen dada. Con esto generamos una función a la cual podemos indicarle cualquier parámetro y la cual usaremos para la generación de descriptores. La función es la siguiente

```
def hog(img,gamma=0.3,cell_size=6,block_size =3,normalization_tipe=2):

    # GAMMA/COLOR NORMALIZATION
    #Compute the normalize gamma and colour

    img = np.power(img,gamma, dtype=np.float32)


    # GRADIENT COMPUTATION
    # Compute centered horizontal and vertical gradients with no smoothing
    kernel = np.asarray([-1, 0, 1])
    kernel_vacio = np.asarray([1])

    gradientsx = cv2.sepFilter2D(img, -1, kernel,kernel_vacio)
```

```
    #showImgAndWait("Gradientes", gradientsx)

    gradientsy = cv2.sepFilter2D(img, -1,kernel_vacio, kernel)
    #showImgAndWait("Gradientes", gradientsy)



    # SPATIAL / ORIENTATION BINNING
    # Calculate the magnitudes and angles of the gradients
    magnitude, angle = cv2.cartToPolar(gradientsx, gradientsy, angleInDegrees=True)

    # If the image is in color, we pick the channel with the biggest value as the
    # intensity of that pixel.
    intensity = np.argmax(magnitude, axis=2)


    x, y = np.ogrid[:intensity.shape[0], :intensity.shape[1]]
    max_angle = angle[x, y, intensity]
    max_magnitude = magnitude[x, y, intensity]

    # 0-360 angle to 0-180
    max_angle = (360 - max_angle) % 180


    #generate the cells
    cells = genCells(max_angle,max_magnitude,cell_size)

    #generate the blocks with normalization and Gaussian
    blocks_normalized = genBlocks(block_size,cells,normalization_tipe=normalization_tipe
)

    #return a vector
    return blocks_normalized.flatten()
```

Clasificador

Ya disponemos de una función que nos permite obtener los descriptores HOG de una imagen. Por tanto, nuestro siguiente objetivo va a ser entrenar un clasificador SVM con las imágenes dadas en la base de datos. La base de datos proporcionada nos ofrece una gran cantidad de imágenes de personas en distintas posiciones y con distintas perspectivas. La base de datos para reducir el tamaño del archivo usa enlaces simbólicos entre las imágenes lo que nos ha dado algunos problemas, de modo que finalmente hemos generado las mismas imágenes con tamaño 128x64 con las figuras humanas.

Para los casos negativos hemos seguido el proceso indicado en la documentación de la base de datos. Se eligen aleatoriamente ventanas de 128x64 de las imágenes negativas. De este modo podemos obtener un gran número de ejemplos negativos.

Es esencial que todas las imágenes tengan la misma dimensión, en este caso seguimos las recomendaciones del paper de usar 128x64. Esto se debe a que los datos del descriptor de cada imagen serán las características con las que entrenaremos nuestro clasificador. Por tanto, todas

las imágenes deberán tener un descriptor de igual tamaño, es decir tener las mismas dimensiones.

Como hemos mencionado anteriormente hemos querido experimentar con varios hiperparámetros del algoritmo HOG, por lo que se ha implementado una función que nos permita indicar los parámetros deseados y que aplique el algoritmo a todas las imágenes de la base de datos, ya sean positivas y negativas. Como resultado tendremos los descriptores de todas las imágenes.

```
1.  #function to generate the
2.  def generateFeatures(name,gamma=0.3,cell_size=6,block_size =3,normalization_tipe=2):
3.      images_pos = loadImgs("./data/images/pos_person")
4.      images_neg = loadImgs("./data/images/neg_person")
5.
6.      #calculate de descriptor and save the data
7.      pos_data = np.array([hog(img,gamma,cell_size,block_size,normalization_tipe) for img
    in images_pos])
8.      pos_data = pd.DataFrame(pos_data)
9.      pos_data.to_csv("./data/descriptor/" +name +"_pos.dat", sep=" ",index=False,header=F
    alse)
10.
11.     #calculate the descriptor negative and save it
12.     neg_data = np.array([hog(img,gamma,cell_size,block_size,normalization_tipe) for img
    in images_neg])
13.     neg_data = pd.DataFrame(neg_data)
14.     neg_data.to_csv("./data/descriptor/"+name +"_neg.dat", sep=" ",index=False,header=Fa
    lse)
```

Como podemos ver con esta función cargamos todas las imágenes y aplicamos nuestra función hog con los parámetros indicados. Con esta función tendríamos los datos de casos positivos y negativos, para nuestro clasificador. Se almacenan en archivos para evitar tener que recalcularlos. En nuestra experimentación, que veremos más adelante hemos generado 4 ficheros, uno para cada tipo de normalización.

A continuación, el siguiente paso va a ser el entrenamiento de una SVM. El problema que tenemos es un problema de clasificación ya que debemos determinar si hay o no personas en la imagen, es decir 1 o 0. Para ello hemos definido la siguiente función con parámetros para indicar los ficheros donde queremos almacenar el clasificador, donde se encuentran los datos y donde guardar la imagen además del título para la gráfica de la curva ROC generada. Además, se añade un parámetro para indicar si se quiere realizar validación cruzada.

Las métricas para determinar los resultados obtenidos serán el Accuracy y el área de las curvas ROC. La grafica de la curva ROC se almacena en imagen y el resultado se muestra por pantalla.

```
1.  #function to train a SVM
2.  def training(filename_svm,filename_dat,filename_roc,tittle_roc,kernel='rbf',test_size=0
    .2, cv=False):
3.      #load the positive and negative data
```

```
    pos_data = pd.read_table("./data/descriptor/"+filename_dat+"_pos.dat", sep=" ", header= None)
    neg_data = pd.read_table("./data/descriptor/"+filename_dat+"_neg.dat", sep=" ", header=None)


    #add the label
    pos_data["etiqueta"] = np.ones(pos_data.shape[0],dtype=np.uint8)
    neg_data["etiqueta"] = np.zeros(neg_data.shape[0],dtype=np.uint8)

    #if we want do cross-validation
    if(cv):
        datos = pd.concat([pos_data,neg_data])
        x_datos = datos.drop('etiqueta', axis=1)
        y_datos = datos['etiqueta']
        #realizando validación cruzada 5-cross validation
        svclassifier2 = SVC(kernel=kernel)
        scores = cross_val_score(svclassifier2, x_datos, y_datos, cv=5)
        # exactitud media con intervalo de confianza del 95%
        print("Accuracy 5-cross validation: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
    else:
        #shuffle default
        train_pos, test_pos = train_test_split(pos_data, test_size = test_size, random_state=50)
        train_neg, test_neg = train_test_split(neg_data, test_size = test_size, random_state=50)

        train = pd.concat([train_pos,train_neg])
        test = pd.concat([test_pos,test_neg])

        x_train = train.drop('etiqueta', axis=1)
        y_train = train['etiqueta']

        x_test = test.drop('etiqueta', axis=1)
        y_test = test['etiqueta']

        svclassifier = SVC(kernel=kernel, gamma= 3e-2)
        svclassifier.fit(x_train, y_train)

        pkl.dump(svclassifier, open("./data/models/" + filename_svm +".p", "wb"))
        # make the predict
        y_pred = svclassifier.predict(x_test)
        acc_test=accuracy_score(y_test, y_pred)
        print("Acc_test: (TP+TN)/(T+P)  %0.2f" % acc_test)

        ##curve ROC
        fpr, tpr, thresholds = roc_curve(y_test, y_pred)
        roc_auc = auc(fpr, tpr)

        plt.figure()
        patch = mpatches.Patch(color='blue', label='ROC curve. area = {}, error = {}'.format(np.round(roc_auc, 4),

np.round(1 - roc_auc, 4)))
        plt.legend(handles=[patch], loc='lower right')
        plt.plot(fpr, tpr, color='blue')
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
```

```
58.         plt.title(tittle_roc)
59.         plt.savefig("./result/"+filename_roc, dpi=700)
60.         #plt.show()
61.         plt.clf()
```

La función es simple, y sigue los siguientes pasos. En primer lugar, se cargar los datos y se le añaden las etiquetas correspondientes. A continuación, diferenciamos si se quiere realizar validación cruzada o un test simple. Si se trata de validación cruzada, la cual se usa para asegurarnos que los resultados no están sesgados debido a un determinado reparto entre test y train, usamos las funciones que nos proporciona el módulo ***sklearn*** y se muestran los resultados.

En el caso de un test simple dividimos el conjunto de datos reservando un 20% de los datos para test. Se mantiene la proporción de datos negativos y positivos por lo que la separación se realiza de forma separada. Finalmente entrenamos nuestro clasificador y realizamos el test. Obtenemos las métricas de Accuracy y la curva ROC que nos parecen una buena forma de valorar los resultados de nuestro clasificador.

Con esta función podremos generar clasificador y valorar los efectos en los cambios de parámetros y decisiones en cuanto al diseño.

Experimentación

Existen múltiples parámetros y factores que pueden influir en los resultados finales de nuestro algoritmo. Hemos decidido experimentar con alguno de ellos, aunque otros hemos usado los recomendados por los autores. Por ejemplo, no hemos experimentado con el número de orientaciones, el solapamiento o el tamaño de la ventana. Consideramos que son parámetros que los autores dan los valores óptimos y con cierta diferenciación con el resto de opciones.

Sin embargo, hemos decidido realizar experimentación con el parámetro de la corrección gamma, el cual no se comentan los efectos. También con la normalización el cual destacan que es uno de los factores más relevantes, y con la aplicación de la máscara Gaussiana. Por último, hemos probado dos kernel para el entrenamiento de SVM. Para la comprobación de la normalización y los distintos kernel hemos probado ambas cosas conjuntamente para comprobar todas las posibles combinaciones y ver los resultados.

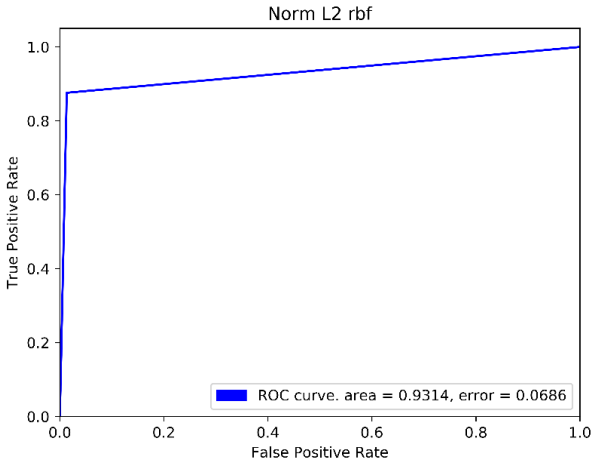
Corrección gamma

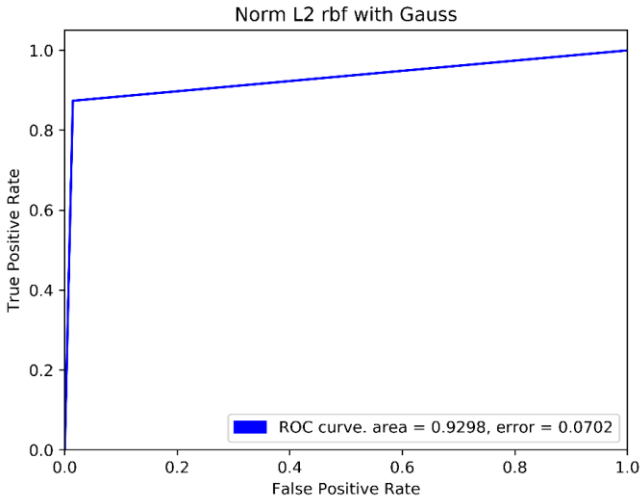
En primer lugar, respecto a la corrección gamma hemos probado varios valores en el rango 0.2-0.6 entre los cuales no hemos apreciado una diferencia clara en los resultados por lo que hemos decidido usar finalmente 0.3. Como imaginábamos es un parámetro que no afecta muy fuertemente en los resultados

Mascara gaussiana

El siguiente parámetro que hemos experimentado ha sido el uso de una máscara gaussiana a la hora de normalizar los bloques, para remarcar que los histogramas más al borde tengan una

menor influencia. Para comparar hemos usado un kernel Gaussiano y normalización L2 ya que según el artículo no existe una gran diferencia con el resto de normalizaciones y los resultados obtenidos son los siguientes





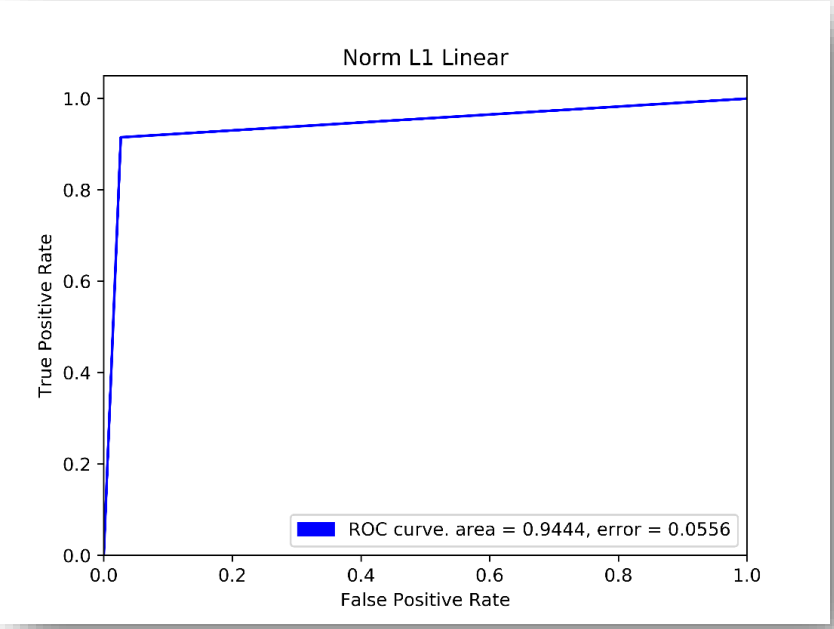
Accuracy para rbf-L2 sin mascara Gaussiana = 0.95
Accuracy para rbf-L2 con mascara Gaussiana = 0.94

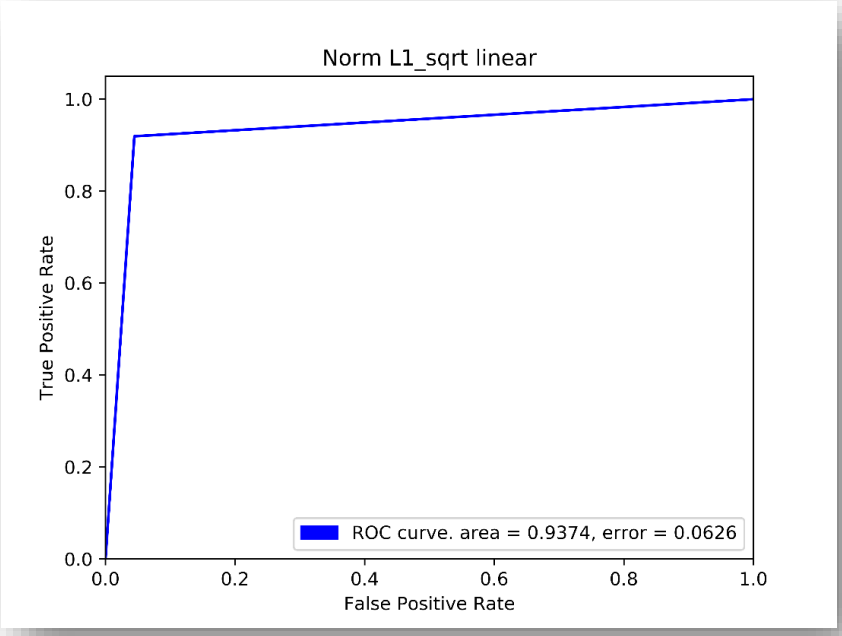
Como podemos ver los resultados son muy similares y el uso de la máscara gaussiana no aporta ninguna mejora por lo que no se aplicará y ahorraremos tiempo de computo en el descriptor hog, lo cual nos servirá para realizar una aplicación en tiempo real.

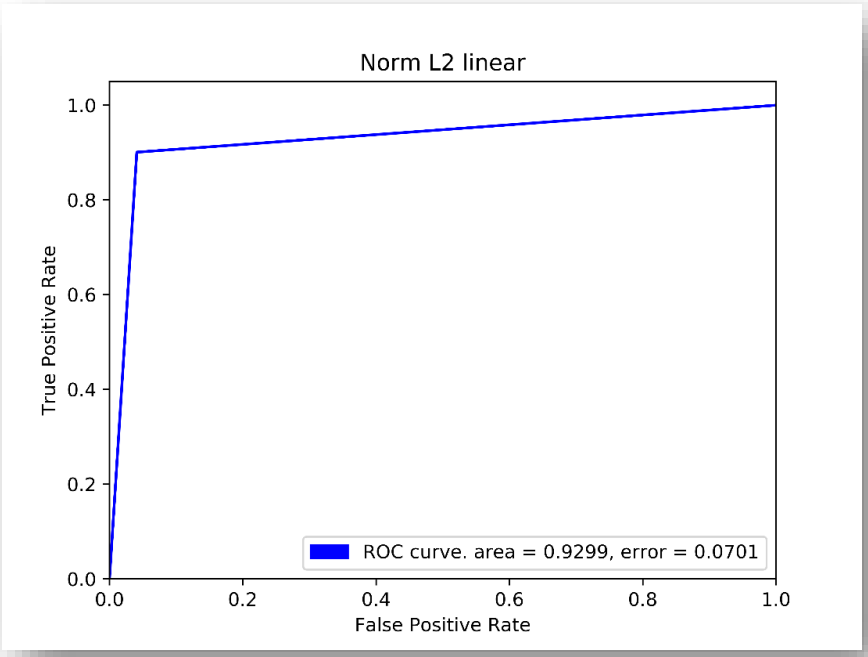
Normalización y kernel

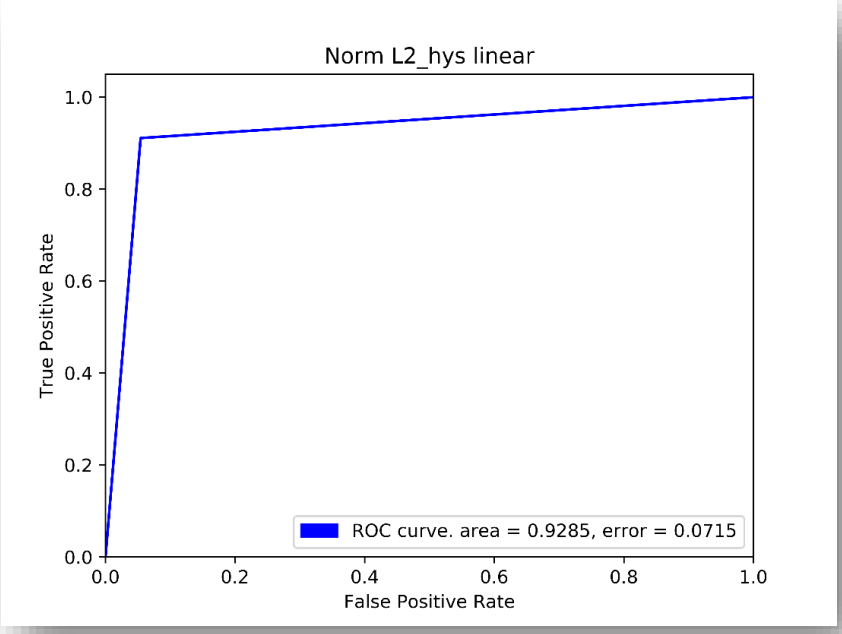
En este apartado hemos querido experimentar con el factor que los autores indican como más relevante y con el núcleo que usará el clasificador. Para ello hemos probado con todas las posibles combinaciones. Para el kernel radial se ha usado $\gamma = 3e - 2$. Veamos primero los

resultados para un kernel lineal que es el que recomiendan por defecto con los 4 tipos de normalizaciones:









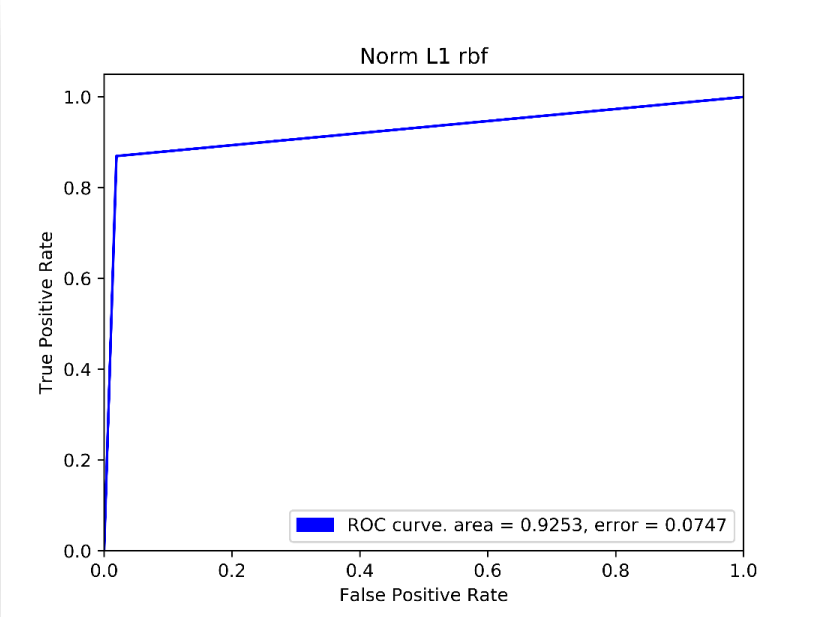
Para normalización L1 ->Acc_test: (TP+TN)/(T+P) 0.95

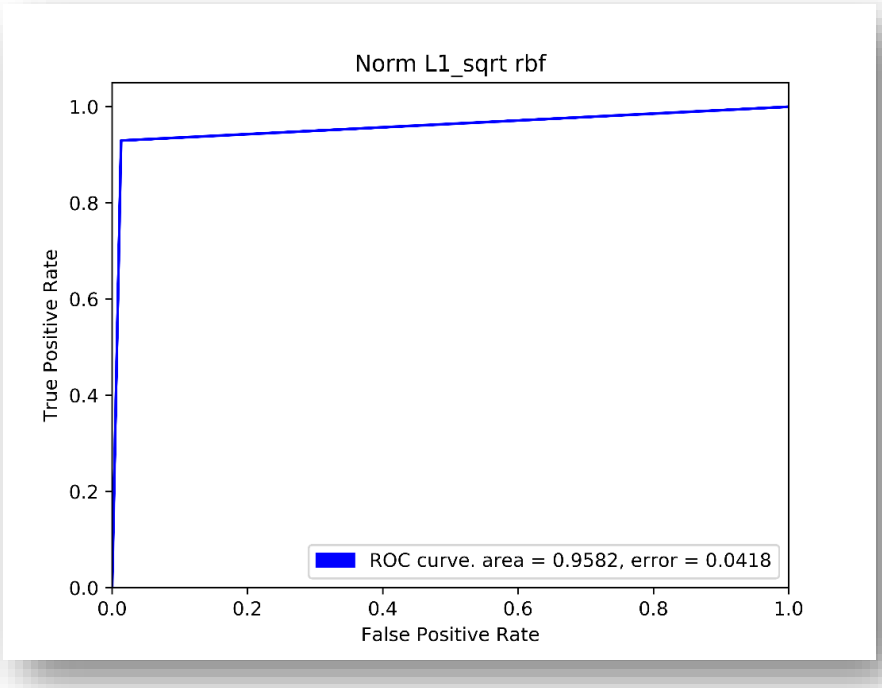
Para normalización L1 sqrt ->Acc_test: (TP+TN)/(T+P) 0.94

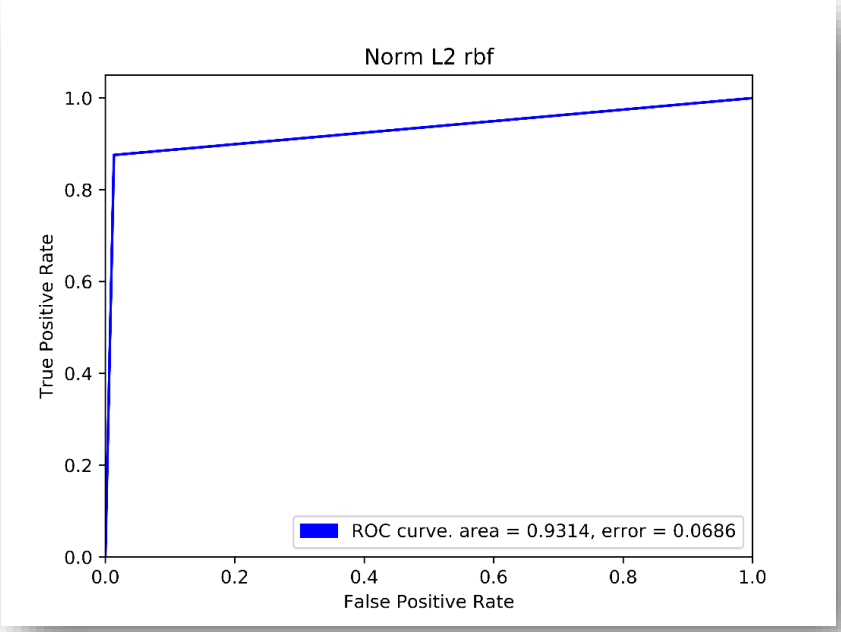
Para normalización L2 ->Acc_test: (TP+TN)/(T+P) 0.94

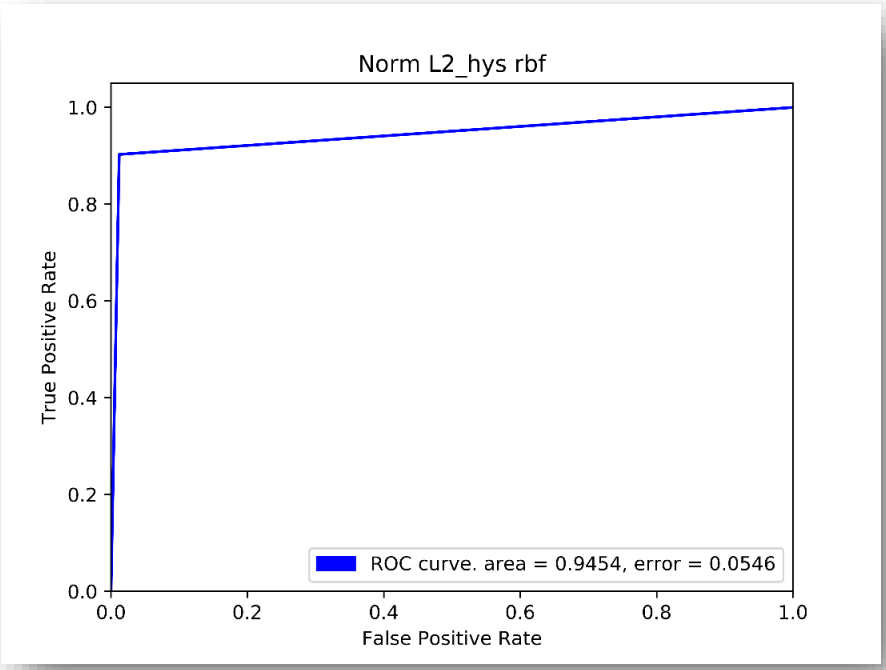
Para normalización L2-hys ->Acc_test: (TP+TN)/(T+P) 0.93

Como podemos ver los resultados son bastante buenos y tenemos un área por debajo de la curva ROC que nos indica que nuestros clasificadores son bastante fiables. En el caso del kernel lineal no se observa una gran diferencia entre los 4 tipos de normalizaciones. Vamos a ver si el kernel gaussiano obtiene mejores resultados como apuntaba el artículo.




Norm L1_sqrt rbf
True Positive Rate
False Positive Rate
ROC curve. area = 0.9582, error = 0.0418
0.0
0.2
0.4
0.6
0.8
1.0


Norm L2 rbf
True Positive Rate
False Positive Rate
ROC curve. area = 0.9314, error = 0.0686
0.0
0.2
0.4
0.6
0.8
1.0



Para normalización L1 ->Acc_test: (TP+TN)/(T+P) 0.94

Para normalización L1 sqrt ->Acc_test: (TP+TN)/(T+P) 0.97

Para normalización L2 ->Acc_test: (TP+TN)/(T+P) 0.95

Para normalización L2-hys ->Acc_test: (TP+TN)/(T+P) 0.96

Como podemos observar en este caso si existe una pequeña mejora con respecto al kernel lineal en todos los tipos de normalización. Si observamos los datos que nos ofrece la curva ROC podemos ver claramente que el área por debajo de la curva es muy alta. También es cierto que con el kernel gaussiano la normalización L1 obtiene peores resultados que las demás. Por lo que en nuestro caso finalmente nos decantamos por usar un kernel gaussiano con normalización L2 que es la que indican por defecto los autores. Para asegurarnos de que nuestra elección es correcta y que los resultados obtenidos están sesgados por una distribución train-test determinada realizamos validación cruzada para asegurarnos que nuestro clasificador se comporta correctamente ante diferentes datos de train

Accuracy 5-cross validation: 0.95 (+/- 0.01)

Como podemos ver los resultados son más que favorables y este definitivamente será el clasificador que usemos. Para asegurarnos de que nuestro clasificador esté preparado con un mayor número de datos, lo que haremos finalmente será entrenarlo con todos los datos, si hacer división para test.

Aplicación a problema real

Ahora vamos a desarrollar una función que realmente pueda ser usada por un usuario final, el cual solo debería introducir una imagen en la que quiere detectar los peatones.

El problema en este punto se recrudece, ya que nuestro clasificador acepta imágenes de 128x64 y ahora queremos ofrecer que funcione para cualquier imagen. ¿Pero cómo sabemos dónde se encuentran los peatones? Debemos recorrer la imagen con una ventana deslizante y recorrer todos los patches de 128x64 para detectar los peatones.

Hasta este punto podemos pensar que esto tiene un coste computacional alto, ya que cómo determinamos el desplazamiento de la ventana. Si nos movemos de un pixel en pixel, el coste computacional se vuelve intratable. Si nos desplazamos demasiados pixeles, podemos perdernos información y no detectar peatones.

Como podemos ver es una decisión difícil, pero el problema empeora ya que tenemos que pensar que podemos tener peatones a distintas distancias. Algunos más cercanos no entran completamente en nuestra ventana. Por lo que debemos realizar una pirámide gaussiana de la imagen original y aplicar a cada nivel la ventana deslizante. Como podemos ver esto se está convirtiendo en un problema terriblemente costoso.

Más adelante veremos las consecuencias de estas decisiones y comentaremos los resultados.

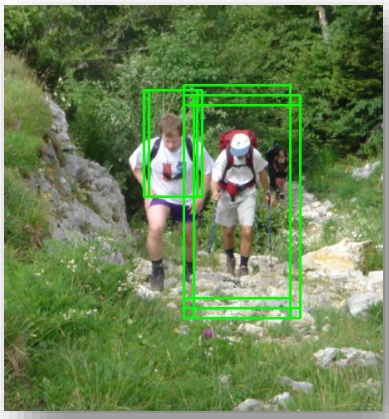
La función desarrolla que realiza este proceso y pinta un recuadro en la región donde se detectó el peatón es la siguiente:

```
1.  #function to the pedestrian detection
2.  def pedestrian_detection(img,file_classifier, hor_jump=4,vert_jump=4):
3.
4.      result_img = np.copy(img)
5.      pyramid = [img]
6.      new_img = img
7.      #generate the  Gaussian pyramid
8.      while np.shape(new_img)[0] >= 128 and np.shape(new_img)[1] >= 64:
9.          new_img = cv2.pyrDown(new_img)
10.         pyramid.append(new_img)
11.
12.     #load the classifier
13.     svm = pkl.load(open(file_classifier,"rb"))
14.     for level, img_pyramid in zip(range(len(pyramid)), pyramid):
15.         for i in range(0, np.shape(img_pyramid)[0] - 128, vert_jump):
16.             for j in range(0, np.shape(img_pyramid)[1] - 64, hor_jump):
17.
18.                 sub_img = img_pyramid[i:i + 128, j:j + 64]
19.                 dst = hog(sub_img, 0.3)
20.                 prediction = svm.predict(dst.reshape(-1, dst.shape[0]))
21.                 #detection of pedestrian
22.                 if prediction[0] == 1.0:
23.                     upper_left =((j* 2**level,i * 2**level))
24.                     down_right = ( (j+64) * 2**level,(i+128) *2**level )
25.                     cv2.rectangle(result_img,upper_left,down_right,(0,255,0),thickness=1
    )
```

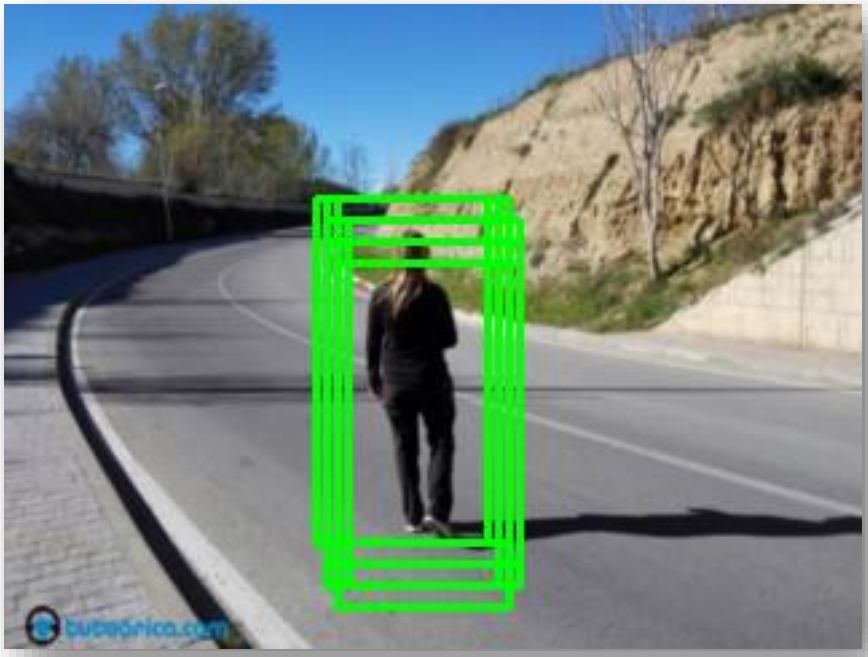
```
26.     return result_img
```

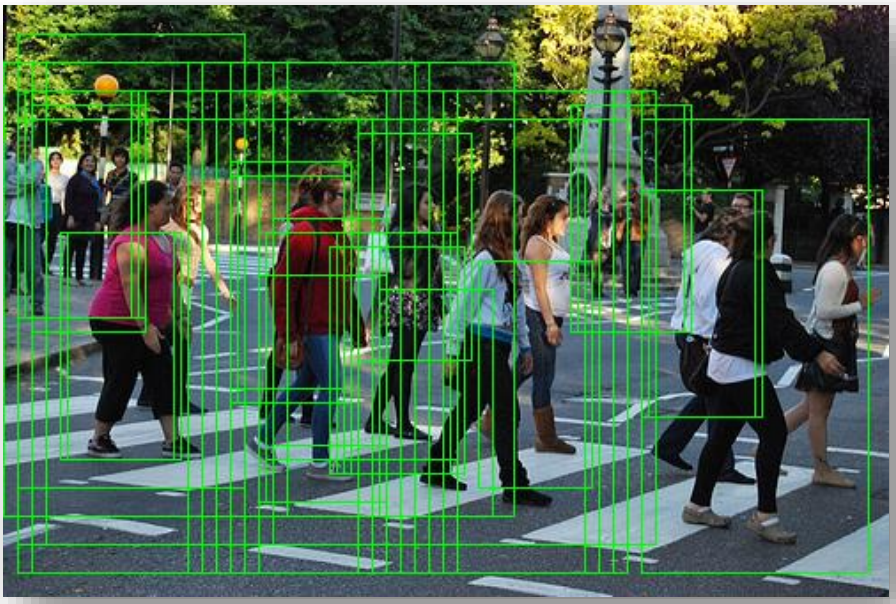
Valoración de los resultados

Hemos probado con varias imágenes escogidas al azar para comprobar los resultados de nuestro detector de peatones. En primer lugar, vamos a probar con una imagen, de las que usamos como prueba para asegurarnos que nuestra aplicación funciona con imágenes para las cuales no debería dar una respuesta sin error.

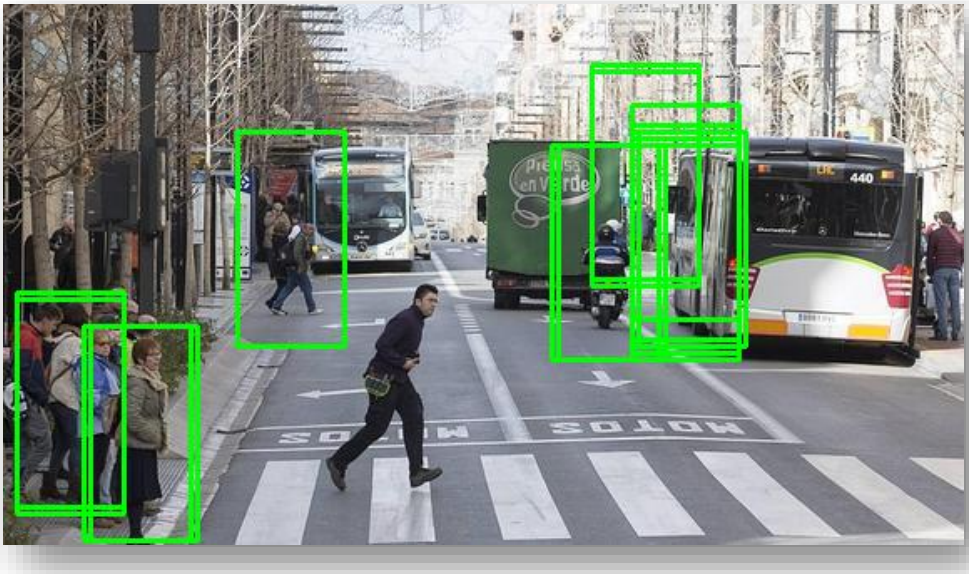


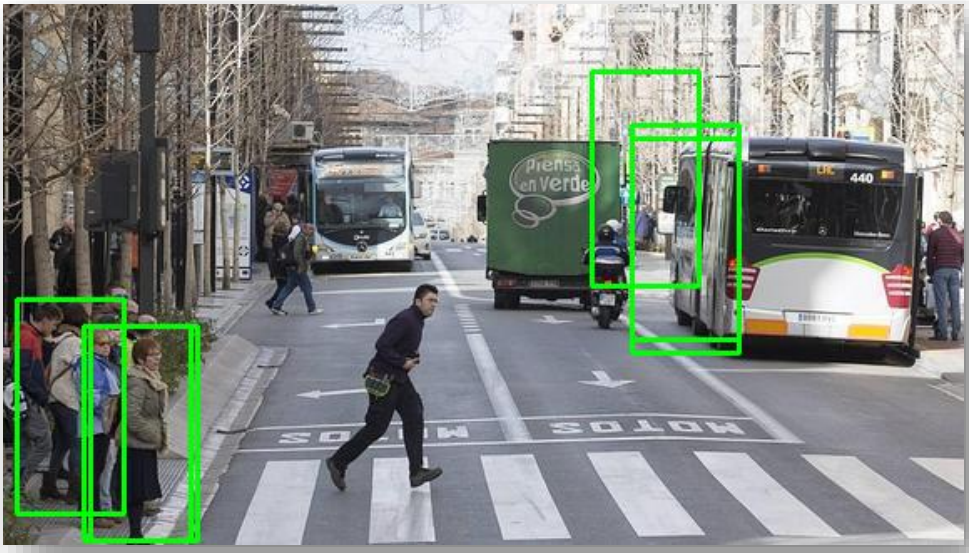
Como podemos ver los resultados son bastante satisfactorios y se detectan correctamente los peatones. A continuación, vamos a probar con varias imágenes ajenas al data set y que prueben que nuestro método funciona correctamente:





En estas imágenes podemos ver que nuestro método da unos resultados muy buenos y que realmente sirve para el objetivo que nos marcábamos. Vamos a ver ejemplos donde nuestro método puede sufrir error y que realmente no consigue un resultado bueno.





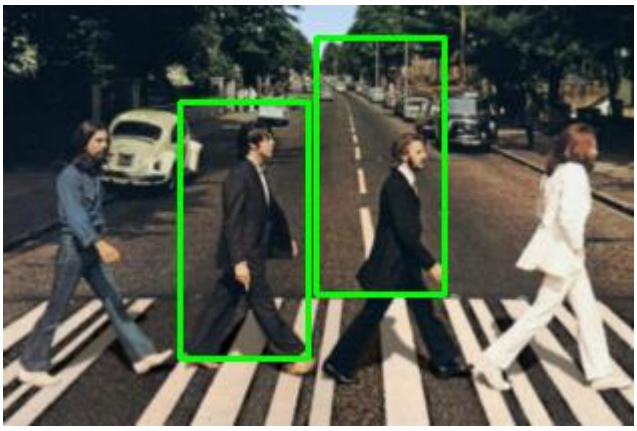
Aquí tenemos dos imágenes en las que podemos ver como en la primera se han detectado más peatones, como por ejemplo los que están cruzando al fondo. La diferencia se debe a que en la segunda imagen se ha usado un mayor salto para la ventana deslizante y esto ha supuesto que nos saltemos información. En la primera se ha usado saltos de 4 pixeles en ambos ejes y en la segunda 4 en el eje x y 8 en el eje y. Como podemos ver no hay gran diferencia, pero ha puesto de

manifiesto que nuestro método es sensible a como movemos la ventana deslizante ya que en la segunda no se han detectado algunos peatones.

En ambas no se ha detectado al peatón que está más cercano cruzando. Este se debe en parte al mismo motivo. Ya que es una figura mayor la cual puede que en el primer nivel de la pirámide no se detecte. ¿Pero no habíamos añadido una pirámide para poder detectarlo? El problema es que en niveles más bajos los saltos son más acentuados y por tanto puede darse con facilidad que la ventana no capte de forma correcta el peatón, de forma que en una iteración captara solo parte y al saltar captara otra pequeña parte, no detectando correctamente la existencia de un peatón.

Este problema se podría solucionar si no hiciéramos saltos y recorriéramos con la ventana todos los pixeles a todos los niveles. El problema de esto es el coste computacional, ya que para cada ventana calculamos el descriptor y clasificamos. Para obtener estos resultados se han usado imágenes de resolución baja y el tiempo supera los 5 minutos. Si recorremos todas las casillas el tiempo hace que nuestra aplicación necesite posiblemente más de una hora para una ejecución.

Veamos un último ejemplo que hace esto aún más visible en la famosa imagen de los Beatles cruzando la calle.



Como podemos ver se han detectado las dos figuras centrales y sin embrago las figuras de los extremos no. Esto se debe a lo mencionado anteriormente y es que son figuras muy grandes, por lo que se deben detectar en niveles profundos de la pirámide. En estos niveles un salto de 4 pixeles puede suponer perder la figura y que no aparezca centrada en la ventana, provocando un error.

Conclusiones.

Como conclusión final podemos decir que el método nos ha dejado un mal sabor de boca, ya que hoy en día, si la visión por computador no puede aplicarse en tiempo real, es difícil darle una utilidad.

Nuestras expectativas eran poder usar este método para detectar peatones en tiempo real y poder abordar el problema desde el punto de vista del automovilismo, de modo que con esta técnica se pudiera dar una solución a detectar peatones. Nuestra idea era usar la detección para determinar si existen peatones y que se pudiera usar en un sistema de decisión que determinara si el peatón se encuentra cerca o lejos, haciendo uso del nivel de la pirámide en el cual fue detectado. Además, podría determinarse si se encuentra en la zona izquierda, central o derecha del coche. Con estos datos se podría generar un sistema de decisión que controlara el coche y lo hiciera esquivar peatones o frenar a tiempo.

La realidad es que este método tiene cierta fiabilidad, pero computacionalmente es imposible llevarlo a tiempo real. Ya que, aunque redujéramos el cálculo del descriptor y las características con las que entrena el clasificador, por ejemplo, eliminando solapamiento, con lo que se perdería robustez, el verdadero problema viene al tener que pasar la ventana deslizante y usar la pirámide gaussiana. Otra posible forma para reducir el tiempo, sería usar paralelismo y usar varias hebras para pasar la ventana deslizante, pero aun así no se alcanzaría que el proceso tardara un segundo, aunque usáramos un procesador de última generación.

Por tanto, debemos de decir como conclusión que es una técnica interesante pero hoy en día no tendría una aplicación real ya que la ejecución en tiempo real es casi un requisito obligatorio. Además, para la aplicación en el mundo del automóvil que queríamos darle, la velocidad de decisión sería vital.

References

Dalal, N. & Triggs, B., n.d. Histograms of Oriented Gradients for Human Detection. *INRIA Rhone-Alps, ˆ 655 avenue de l'Europe, Montbonnot 38334, France.*