Visión por Computador (2017-2018)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Proyecto:
Detección de peatones en imágenes

Mª del Mar Alguacil Camarero

Índice

1. Introducción

Este proyecto se centra en la detección de peatones, ampliamente estudiado en los últimos años debido, principalmente, a su gran a aplicación en el campo automovilístico ya que se pretende evitar accidentes de tráfico producidos por errores humanos debidos a distracciones o pérdida de control del vehículo que pueden dar lugar a una colisión con graves efectos tanto para las personas tripulantes del vehículo como para las personas localizadas en las cercanías del mismo.

Para ello, a lo largo de este informe se introducirán, definirán y explicarán algunos de los métodos que usa la visión por computador para resolver este problema, con los cuales se construirá un programa para poder ver su efecto en imágenes reales.

2. Problema a resolver

En este proyecto, haremos una simplicación del problema ya que sólo detectaremos un peatón en una imagen estática dada. Para resolver este debemos tener en cuenta algunas consideraciones:

- Entenderemos por fondo todo aquello que, pertenezca a la imagen, pero que no sea una persona.
- Dispondremos de una base de datos de tamaño considerable con imágenes de un tamaño fijo (en nuestro caso, 64×128) dividas en dos carpetas: una de ellas contendrá fotos de personas en distintas posiciones, y otra serán todo fondo. Esta base de datos nos servirá para entrenar nuestro modelo, y poder determinar si esta contiene o no a una persona.

En la resolución se empleará el descriptor HOG para extraer las características de las distintas imágenes o subimágenes del tamaño prefijado, y a partir de estas, se empleará el clasificador SVM entrenado con las características extraidas de HOG de las fotografías de la base de datos, el cual nos permitirá determinar (con una cierta probabilidad) si aparece un peatón, o no, en una determinada imagen.

Para comprobar la eficacia de todo esto, se calculará el porcentaje de acierto del conjunto de entrenamiento y de prueba. Y, por último, como aplicación práctica, se muestran distintas imágenes de tamaño variable en las cuales se emplea el método de ventana deslizante (con una ventana de tamaño fijo: 64×128) para buscar si hay peatones o no, en la imagen.

3. Conceptos previos

En esta sección, se explicarán de forma teórica las distintas herramientas y pasos que se seguirá en el desarrollo de la práctica, para una mejor comprensión de la implementación realizada en el problema de la detección de peatones.

Como sabemos el esquema general de un detector de objetos es:

1. Generación de candidatos
2. Clasificación de candidatos
3. Refinamiento de la decisión

Para la clasificación de candidatos intervienen principalmente dos componentes. Por un lado, un descriptor que nos permite representar el contenido de una ventana en forma de vector, y un clasificador que nos va a permitir determinar cómo se fija la frontera en el espacio del descriptor entre las ventanas que corresponden al objeto (en nuestro caso, al peatón) y las ventanas que corresponden al fondo.

Como descriptor del contenido de las ventanas vamos a utilizar HOG (*Histogram of Oriented Gradients*), y como método de clasificación SVM (*Support Vector Machine*) que es un clasificador bastante eficiente y utilizado. A continuación desarrollamos ambos algoritmos de forma teórica aunque sólo se implementará HOG siguiendo los distintos pasos, mientras que para SVM nos ayudaremos de las herramientas que nos brinda *Python*.

3.1. Descriptor: HOG

HOG es un descriptor de la imagen que utiliza el gradiente en cada uno de los píxeles como información básica. Definiendo el gradiente como un cambio de intensidad de la imagen en una cierta dirección, aquella dirección en la que el cambio de intensidad es máximo. Además, este se calcula para cada uno de los píxeles de la imagen y queda definido por dos valores:

- Dirección donde el cambio de intensidad es máximo.
- Magnitud del cambio de la dirección de máxima variación.

Así para cada píxel estos dos valores nos permiten distinguir distintas situaciones de la configuración local alrededor del píxel en relación al cambio de contraste y a la forma local. Representamos el gradiente como un vector con dirección en la orientación de máximo cambio y una longitud proporcional a la magnitud del gradiente.

El gradiente se puede calcular de varias formas diferentes. Pero en el contexto del descriptor HOG este cálculo se realiza a partir de la diferencia de intensidad de los píxeles vecinos en dirección tanto horizontal como vertical. Por ejemplo, si tenemos una imagen I representada en escala de grises (valores de intensidad) y queremos calcular el gradiente en un determinado punto (x, y). Para calcular el gradiente empezamos calculando la diferencia en la dirección horizontal de x que se puede calcular como la diferencia de intensidad entre el píxel situado a la derecha $((x+1, y))$ menos el píxel situado a la izquierda $((x-1, y))$:

$$dx = I(x+1, y) - I(x-1, y)$$

Igualmente podemos calcular la diferencia en la dirección vertical como la diferencia entre el píxel situado arriba y el píxel situado debajo.

$$dy = I(x, y-1) - I(x, y-1)$$

Estas diferencias en horizontal y vertical se pueden calcular para todos los píxeles de la imagen, y cada píxel no dará información local alrededor de este. Esto equivale a convolucionar la imagen con la máscara $[1, 0, -1]$ para obtener dx y su traspuesta para dy.

A partir de estas diferencias podemos calcular la orientación y la magnitud global del gradiente tal y como lo hemos definido antes. Trasladamos los valores de dx y dy al eje de coordenadas, y estos dos valores en el eje de coordenadas nos permiten definir un vector. Dicho vector es el que corresponde al vector gradiente para este píxel. A partir de este podemos calcular la orientación del gradiente en un punto y la magnitud del mismo en ese punto. La orientación se definirá como el ángulo que forma este vector con el eje horizontal:

$$\theta(x, y) = arctan\frac{dy}{dx}$$

Mientras que la magnitud del gradiente es la longitud del vector:

$$g(x, y) = \sqrt{dx^2 + dy^2}$$

Así a partir de las diferencias en horizontal y en vertical, podemos calcular la orientación y la magnitud del gradiente para cada uno de los píxeles de la imagen. Y esta información nos da una información global de la imagen.

En el descriptor HOG el color se trata priorizando aquel color que domina para cada uno de los píxeles. Esto se consigue calculando para cada uno de los píxeles de la imagen el gradiente para uno de los tres canales de color (rojo, verde y azul). Una vez calculado los tres canales por separado, se toma, para cada píxel, aquel canal que tiene una magnitud mayor. De esta forma, estamos priorizando localmente el color que domina alrededor de un determinado píxel.

A continuación veremos como la información local del gradiente a nivel de cada uno de lo píxeles se puede agregar en forma de histogramas calculados en diferentes áreas de la imagen.

El problema con la representación a nivel de píxel del gradiente es que difícilmente se puede utilizar junto con un clasificador en un sistema de detección de objetos, ya que un clasificador, normalmente, requiere como entrada una representación global de toda la imagen de una dimensión fija y, además, esta información a nivel de píxel puede ser muy sensible a pequeñas variaciones tanto en la forma como en la localización del objeto en la imagen. Así pues, se pretende convertir esta información local del gradiente para cada

uno de los píxeles en una representación global de toda la imagen en forma de vector de características. Para ello utilizaremos el descriptor HOG, que es un descriptor global de toda la imagen que captura la forma del objeto.

Para conseguir esta representación global de la imagen, el descriptor HOG actúa en dos pasos diferentes. En primer lugar, divide la imagen en un número fijo de celdas y, para cada una de estas celdas, obtiene un histograma de las orientaciones de los gradientes. En un segundo paso, se calculan los histogramas para todas las celdas, y todos estos histogramas se combinan para obtener la representación global de toda la imagen en forma de vector de características.

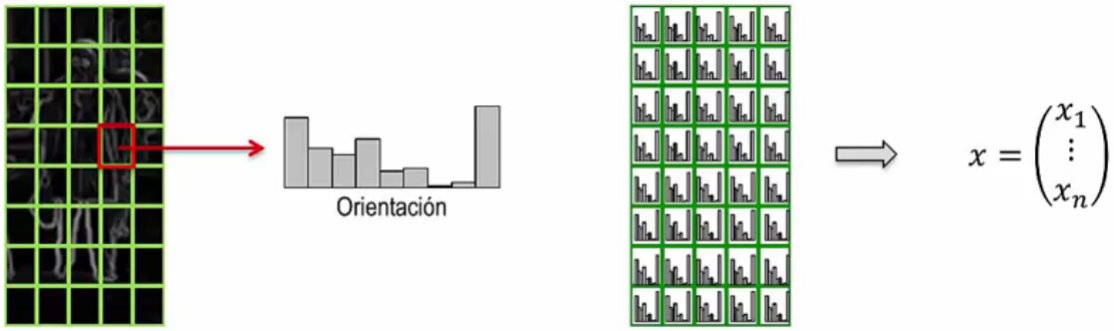

Figura 3.1: Pasos HOG-*Histogram of Oriented Gradients*

A partir de estos dos pasos cada uno de los histogramas de orientación permite capturar información de las orientaciones dominantes en la imagen, que se corresponderán con los valores altos en cada uno de los histogramas. Pero, al mismo tiempo la representación final y al combinar los histogramas de las celdas distribuidas por toda la imagen también captura información espacial de cómo están distribuidos estos gradientes en la imagen.

Primero nos centraremos en el cálculo de los histogramas de orientación para cada una de las celdas, y seguidamente se explicará la agregación de todos estos histogramas para obtener el descriptor final.

El primer parámetro que tenemos que fijar es el tamaño de la celda, valores entre seis y ocho píxeles tanto en ancho como en alto suelen ser valores habituales. Otro aspecto a considerar es cómo se divide el rango de orientaciones en un número de intervalos fijo. La primera consideración afecta al rango de orientaciones en sí mismo ya que podemos tomar la orientación del gradiente con signo, con lo que el rango de orientaciones irá desde 0^{o} hasta 360^{o}, o bien podemos considerar la orientación gradiente sin signo, y de esta forma el rango de orientaciones va desde 0^{o} hasta 180^{o}. Con esta segunda opción, dos gradientes con la misma dirección pero sentidos opuestos se consideran equivalentes y quedan asignados al mismo intervalo. El otro parámetro que hay que fijar es en cuántos

intervalos se divide el rango de orientaciones, siendo la más habitual en 9 intervalos. De esta forma, cada intervalo agrupa un rango de orientaciones de 20º. Fijado los parámetros para el cálculo del histograma, cada uno de los gradientes de la celda, con su orientación y magnitud, quedará asignado a uno de los intervalos en función de la orientación del gradiente. De esta forma, cada intervalo tiene asignados todos aquellos gradientes con una orientación dentro de los límites de ese intervalo. Finalmente, el valor de uno de estos intervalos en el histograma final se obtiene acumulando la magnitud de todos estos gradientes asignados al intervalo. Este cálculo del histograma se puede formalizar matemáticamente a partir de la siguiente expresión:

$$h(k) = \sum_{(x,y)\in C} \omega_k(x,y) g(x,y)$$

Así, si tenemos una celda C que contiene un conjunto de píxeles, cada uno de ellos con un gradiente, con una orientación y una magnitud, para calcular el valor del histograma en una determinada posición k se obtiene como la suma de la magnitud de los gradientes para todos los píxeles y todos los gradientes de la celda ponderado por un factor que nos determina la asociación del gradiente a dicho intervalo k. De forma simplificada, este factor de asignación se definirá de forma que para todos los gradientes cuya orientación esté dentro de los límites definidos por el intervalo valga uno, mientras que para todas aquellas orientaciones que estén fuera de los límites del intervalo, valga cero.

$$\omega_k(x,y) = \begin{cases} 1 & \text{si } (k-1)\delta\theta \leq \theta(x,y) < k\delta\theta \\ 0 & \text{en caso contrario} \end{cases}$$

Este modelo de cálculo de los histogramas de orientaciones parte del principio de asignar cada gradiente a un único intervalo. Es un modelo simple que puede presentar ciertos problemas como puede ser que los gradientes con orientaciones muy similares pueden asignarse a intervalos diferentes. Esto nos puede provocar que pequeñas variaciones en la imagen de entrada acaben provocando variaciones significativas en la representación final. La solución que se propone consiste en asignar cada píxel a los dos intervalos más cercanos, con un peso proporcional a la distancia de la orientación del gradiente a cada uno de los intervalos. Así pues, para asignar un gradiente a un determinado intervalo k calcularemos la distancia entre la orientación del gradiente y la orientación del centro del intervalo $(\theta(x,y) - \theta_k)$. Y esta distancia se utilizará para calcular el factor de asociación de un determinado gradiente al intervalo, y se normaliza por el rango de cada uno de los intervalos del histograma.

$$\omega_k(x,y) = max\left(0, 1 - \frac{|\theta(x,y) - \theta_k|}{\delta\theta}\right)$$

De esta forma, si la distancia es cercana a cero, el factor de asignación será cercano a uno, mientras que si la distancia es cercana al rango del intervalo, el factor de asignación se acercará a cero. Por otro lado, la utilización del máximo en la fórmula hace que para distancias que sean mayores al rango del intervalo, el factor de ponderación sea cero.

Finalmente, este factor de ponderación se utiliza, igual que antes, para acumular la magnitud de todos los gradientes de la celda, de manera que se obtiene el valor final para cada uno de las posiciones del histograma.
Con esta nueva formulación, conseguimos que cada gradiente contribuya no a un único intervalo sino también a los dos intervalos más cercanos en función de su orientación.

El cálculo del histograma se repite para todas las celdas en que se divide la imagen, de forma que para cada una de las celdas tenemos su correspondiente histograma, y en este histograma acumulamos, para cada uno de los intervalos, la magnitud de los gradientes ponderados por el factor de asignación de todos aquellos gradientes que están dentro de la celda.

Sin embargo, este esquema en que cada gradiente contribuye únicamente al histograma de la celda a la que pertenece, puede presentar el mismo tipo de problemas que ya hemos comentado anteriormente para la asignación de las orientaciones, es decir, dos píxeles muy cercanos en la imagen pueden asignarse a celdas diferentes. Por lo tanto, pequeños cambios en la forma o en la localización del objeto pueden dar lugar a variaciones significativas en la representación final ya que puntos que son equivalentes en el contorno del objeto contribuirán a histogramas diferentes en imágenes diferentes. Para solucionarlo, se utiliza la misma estrategia de interpolación que hemos introducido para la asignación de las orientaciones. Cada píxel se asignará a las cuatro celdas más cercanas con un peso proporcional a la distancia del píxel al centro de la celda. Así podemos calcular para cada uno de los píxeles, la distancia de este píxel al centro de la celda en dirección x (d_{ij}^x),y en la dirección y (d_{ij}^y). Estas distancias se utilizan para calcular el factor de asignación de cada uno de los píxeles de las celdas normalizadas por la distancias entre los centros de dos celdas, tanto en la dirección x como en la dirección y.

$$\omega_{ij}^x(x,y) = max\left(0, 1 - \frac{d_{ij}^x}{\delta x}\right) \quad \omega_{ij}^y(x,y) = max\left(0, 1 - \frac{d_{ij}^y}{\delta y}\right)$$

Estos factores de ponderación de asignación de un píxel a una celda en las direcciones x e y, se combinan con el factor de asignación de cada gradiente a uno de los intervalos del histograma que hemos explicado anteriormente, y de esta forma obtenemos el peso final que se aplica a cada una de las magnitudes de los gradientes de todos los píxeles de la imagen para acabar obteniendo el valor concreto del histograma para cada uno de los intervalos.

$$h_{ij}(k) = \sum_{(x,y)} \omega_{ij}^x(x,y)\omega_{ij}^y(x,y)\omega_k(x,y)g(x,y)$$

Por último, veamos cómo se obtiene el vector con la representación final del descriptor a partir de los histogramas de orientación calculados en cada una de las celdas en que se divide la imagen. Esta representación final se va a basar en normalizar y agrupar estos histogramas en forma de bloques.

El punto de partida para obtener la representación final de descriptor son los histogramas de orientación del gradiente, que se calculan para cada una de las celdas en las que se divide la imagen. Uno de los objetivos de cualquier descriptor debe ser conseguir la máxima invarianza posible a todas aquellas variaciones que se pueden producir en la imagen de entrada, ya sean de iluminación, posición, escala, aspecto, etc. Como explicamos anteriormente, al agrupar los gradientes en los histogramas se siguen ciertas estrategias que permiten obtener invarianza a algunas variaciones en la localización y el aspecto de los objetos en la imagen. Sin embargo, cuando tenemos cambios de iluminación, la intensidad del gradiente cambia. Por lo tanto, estos cambios también se van a reflejar en los valores del histograma. Para minimizar estas diferencias en la descripción de las imágenes, va a ser conveniente normalizar los valores de los histogramas, con el objetivo de conseguir que la magnitud global del gradiente sea similar en ambas imágenes. Pero como los cambios de iluminación puede que no sean constantes a lo largo de toda la imagen, no va a ser suficiente aplicar una única normalización uniforme en toda la imagen, sino que va a ser preferible una normalización local adaptada a cada una de las zonas de la imagen. Es por ello que se introduce el concepto de bloque que es una agrupación de varias celdas vecinas. Una configuración bastante habitual en el descriptor HOG es utilizar bloques de dos celdas en horizontal y dos celdas en vertical. Así para cada bloque vamos a coger los histogramas de cada una de las celdas y los vamos a concatenar para obtener el vector con la representación del bloque.

La normalización de los histogramas se va a realizar a nivel de bloque, es decir, vamos a normalizar este vector resultado de concatenar todas las celdas. La normalización se obtiene diviendo cada uno de los componenentes del vector por su norma euclídea, L_2:

$$v = (v_1, ..., v_n) \Rightarrow \|v\|_2 = \sum_{i=1}^{n} v_i^2$$

De esta forma garantizamos que la magnitud global del vector va a ser uno y así vamos a minimizar las diferencias debidas a variaciones locales en el contraste. Se puede introducir en la fórmula una constante ε con un valor muy pequeño para evitar divisiones por cero, en aquellos casos de bloques donde la intensidad sea constante a lo largo del bloque y, por lo tanto, la magnitud total del gradiente sea cero.

$$v' = \frac{v}{\sqrt{\|v\|_2^2 + \varepsilon}}$$

Esta normalización constribuye a reducir las diferencias en la representación final entre imágenes similares y, por lo tanto, obtenemos un descriptor final mucho más robusto.

En la práctica, los bloques se definen de forma que tengan un cierto solapamiento entre ellos. La redundancia que se va a obtener con este solapamiento, va a ayudar a conseguir un descriptor más robusto ante deformaciones y variaciones en la forma del objeto. Habitualmente, los bloques se colocan con una separación de una sola celda entre ellos tanto en horizontal como en vertical, y la representación final del descriptor HOG se

obtiene concatenando la representación normalizada de todos estos bloques solapados. De esta forma, cada celda contribuye a la representación final de varios bloques. Tantos como celdas tengamos en cada uno de los bloques. A pesar de que el histograma básico de la celda va a ser el mismo en todos los bloques, como cada bloque va a aplicar una normalización distinta que depende del resto de las celdas del bloque, la contribución de este histograma a cada uno de los bloques puede ser diferente.

Por lo tanto, en esta representación final del descriptor influyen una serie de parámetros que ya hemos comentado anteriormente. En primer lugar, tenemos el tamaño de cada una de las celdas y el tamaño de la celda junto con el tamaño de la imagen de entrada, nos va a determinar el número de celdas que se pueden colocar en la imagen. Por otro lado, tenemos un par de parámetros que afectan a cómo se construyen los histogramas. Por un lado, si se utiliza el gradiente con o sin signo, y por otro lado, el número de intervalos del histograma. Finalmente, tenemos el parámetro que nos fija cuántas celdas colocamos en cada uno de los bloques. Algunos valores habituales para estos parámetros suelen ser un tamaño de celda de 8×8 píxeles, utilizar el gradiente sin signo, dividir la orientación del gradiente en 9 intervalos y utilizar bloques de 2×2 celdas. Todos estos parámetros nos acaban determinando el número de dimensiones final del descriptor HOG, cuántos componentes tiene el vector (n).

$$hog = (x_1, ..., x_n)$$

Y este número final de dimensiones se puede calcular a partir de la siguiente expresión:

$$n = n^\circ\ bloques \times n^\circ\ celdas/bloque \times n^\circ\ intervalos\ histograma$$

En resumen, HOG:

- Explota la información del gradiente de la imagen (contornos de los objetos), que nos aporta información que puede ser tanto relevante para la detección de los objetos como para el reconocimiento de estos ya que normalmente valores altos se corresponden con los contornos o la silueta de los objetos.
- Permite aprovechar la información, de forma bastante eficiente, del gradiente a partir de combinar histogramas locales que se calculan en celdas de pequeño tamaño que se distribuyen de forma uniforme por toda la imagen. Estos histogramas nos proporcionan información de las orientaciones de los contornos que dominan en cada una de las posiciones de la imagen en sí. Esta información nos permite distinguir la forma de los objetos presentes en una imagen y es una buena base para detectar y reconocer los objetos.
- Los histogramas, que se calculan localmente en diferentes posiciones de la imagen, se agrupan en bloques de un tamaño un poco mayor y estos bloques sirven para normalizar la representación final y para hacerla más invariante a cambios de iluminación y a distorsiones en la imágen. Así la representación final será la concatenación de la representación de todos estos bloques.

3.2. Clasificador: SVM

Como ya sabemos el objetivo de cualquier clasificador es encontrar una frontera que nos permita separar los ejemplos positivos (en este caso, peatones) de los negativos (en este caso, el fondo).

Centrémonos, por tanto, en las máquinas de Vectores de Soporte (SVM) que son sistemas de clasificación binarios, lo que nos permite distinguir entre dos clases. SVM es un clasificador lineal, es decir, la frontera de decisión es un hiperplano W. Este hiperplano W se obtiene de manera que se maximice lo que se denomina el margén entre los ejemplos positivos y los ejemplos negativos, es decir, maximizamos la distancia entre aquellos ejemplos positivos y negativos que están más cercanos entre sí, denominados vectores de soporte. Y el hiperplano que se encuentra busca separa al máximo estos ejemplos de la frontera. De esta forma, estos son los ejemplos que se utilizan para determinar el hiperplano, y por lo tanto, la frontera de decisión se concentra en poder separar aquellos casos más cercanos entre sí, los más difíciles.

Si los datos no son linealmente separables, SVM nos permite aplicar una técnica que se conoce como *kernel trick*, la cual nos posibilita transformar el espacio de característica original en otro espacio de características diferente, y que es linealmente separable, de forma que podremos encontrar la frontera en este nuevo espacio pero sin necesidad de calcular esta transformación de forma explícita.

Idea básica

Esta combinación de descriptor y clasificador junto con el resto de pasos del esquema general de detección, permiten construir un detector de objetos bastante potente que ha sido ampliamente utilizado para detectar tanto peatones como otros tipos de objetos en una imagen y que además ha servido de base para construir y para diseñar otros detectores más especializados.

No existe una única manera de trazar líneas (solución de hiperplanos) que particionen el espacio de características para el problema de dos clases en dos regiones disjuntas siguiendo diferentes criterios. En general, podemos decir que existen dos grandes formas de resolver este problema dentro de lo que es la disciplina del reconocimiento de patrones. Por un lado, tenemos lo que se conoce como modelos generativos que tratan de construir estas fronteras de separación entre clases a partir de estimar la función de densidad de probabilidad que podemos asociar a cada una de las clases. Por ejemplo, los clasificadores de Bayes o el análisis discriminante de Fisher. Por otro lado, tenemos lo que conocemos como modelos discriminativos que tratan de clasificar las muestras sin tener estas funciones de densidad de probabilidad de las clases, y por lo tanto, encuentran la frontera a partir de un conjunto de entrenamiento que sea suficientemente representativo. Por ejemplo, el método de regresión logística, las redes neuronales y el SVM.

Las características principales de los SVM es que son clasificadores lineales cuya solución se basa en encontrar el margen máximo entre las dos clases a partir de unos vectores determinados que es lo que conocemos como vectores de soporte. Al ser un clasificador lineal, la solución del SVM va a ser siempre un hiperplano que permita dividir el espacio de características en dos regiones completamente disjuntas. En el caso particular de dos dimensiones, dos características, esta solución va a ser siempre una línea recta. Para encontrar este hiperplano, se tiene en cuenta sólo un número de muestras limitado del conjunto de entrenamiento, que tendrán unas propiedades concretas. Estas muestras particulares que utiliza este clasificador se les denomina vectores de soporte. Y tendremos vectores de soporte para cada una de las dos clases que tengamos. Los vectores de soporte son elegidos de manera que la distancia (margen entre planos que los contienen) sea máxima. Esta condición implica que lo que buscamos es encontrar la región entre vectores de soporte más amplia posible que esté vacía de muestras de entrenamiento de cualquiera de las dos clases, y el plano intermedio de esta región será la solución. Por lo tanto, podemos decir que la solución del SVM es una solución a un problema de optimización en el que lo que buscamos optimizar es el margen entre las dos clases. Cualquier elección alternativa de los vectores de soporte nos dará una solución alternativa y diferente al hiperplano que permite separar las dos clases. Pero si no es una solución óptima, va a tener un margen más estreho y por lo tanto, vamos a tener un margen de seguridad menor en la clasificación de aquellas muestras más próximas en la frontera entre clases.

La manera que tiene el SVM de encontrar la solución a partir del uso de vectores de soporte proporciona algunas propiedades bastante interesantes. Por un lado, debido a que la solución depende únicamente de la elección de los vectores de soporte, si desplazamos estos vectores, evidentemente, la solución que se encuentra como resultado de la optimación del margen va a cambiar. Sin embargo, debido a que los vectores de soporte son los únicos que contribuyen y, por tanto, las muestras que no son vectores de soporte no intervienen en la definición de frontera, si desplazamos las muestras que no son vectores de soporte la solución del clasificador no va a cambiar, es decir, vamos a obtener el mismo hiperplano como solución. Este hecho proporciona al SVM cierta robustez estadística y reduce el *overfitting*, que implica perder capacidad de generalización cuando el clasificador se concentra demasiado en las muestras de entrenamiento. Por el contrario, SVM generaliza de una manera bastante robusta y no se ve particularmente influido por *outliers* que se encuentren fuera de la zona donde están definidos los vectores de soporte. Este método sólo nos sirve únicamente para clasificar conjuntos que sean linealmente separables y sin solapamiento entre las clases.

Para que el SVM sea eficiente también en problemas donde el conjunto no es linealmente separable existen dos alternativas. Por un lado, se permite cierto solapamiento de clases a partir de relajar la condición del margen (margen suave), y por otro lado, se extiende el modelo de forma que pueda trabajar en conjuntos que no sean linealmente separables mediante la transformación del espacio de características en otro que sí lo es, es lo que se conoce como el truco del kernel o el *kernel trick*.

La relajación de la condición del margen implica que permitimos que algunas muestras estén dentro de la región que nos define el margen entre las dos clases o incluso que queden erróneamente clasificadas. Esto permite que SVM pueda ser robusto al ruido asociado a estas muestras y permite tener cierta tolerancia a errores de clasificación. De forma que podemos gestionar el problema de solapamiento entre clases y podemos trabajar con conjuntos que no sean separables linealmente.

Para conjuntos que no son linealmente separables, lo que hacemos es mapear el conjunto de este espacio de características en los que los datos no son separables linealmente a otro espacio de características de dimensión superior en el que los datos sí que van a ser linealmente separables. Por tanto, en este nuevo espacio de características ya podemos aplicar el SVM de forma estándar. Además con esta transformación entre espacio de características no va a ser necesario encontrarla explícitamente ya que la formulación matemática nos va a facilitar el trabajo y va a permitir definir el producto escalar, y a partir de lo que se conoce como *kernel trick*, únicamente con el producto escalar en este nuevo espacio de características ya vamos a poder encontrar la solución.

Desarrollo matemático

Que el SVM sea un clasificador lineal implica que existe un hiperplano solución, cuya solución se puede expresar de esta de la siguiente forma, a partir de un vector w que es ortogonal al hiperplano solución y un coeficiente de intersección b.

$$w^t x_i + b = 0$$

Este hiperplano solución se obtiene como el hiperplano medio entre otros dos hiperplanos h^+ y h^- que son los que contienen los vectores de soporte de ambas clases respectivamente.

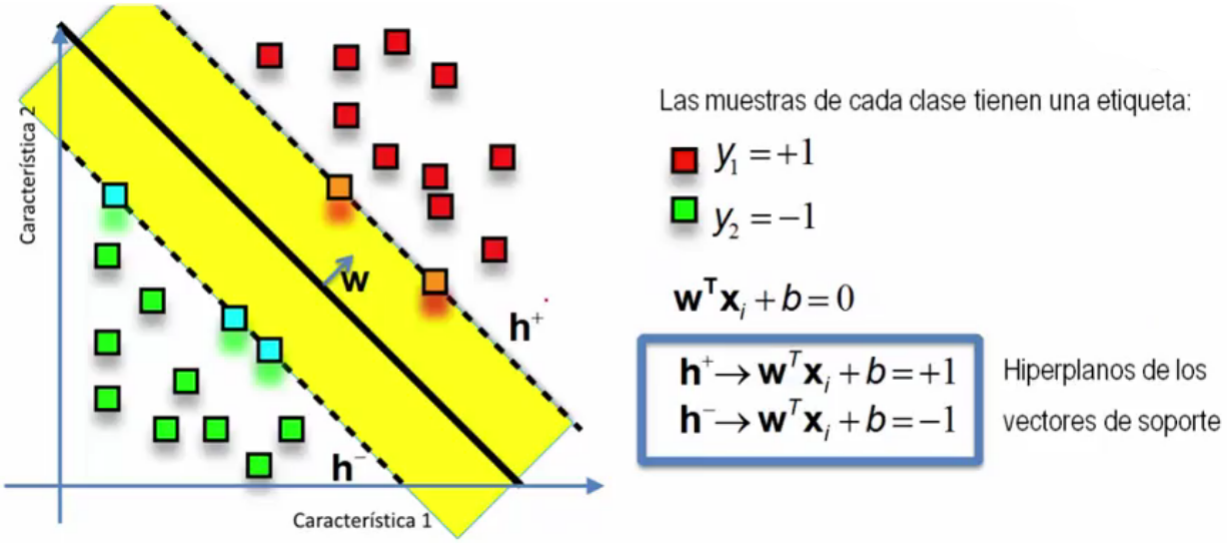

Figura 3.2: SVM-*Support Vector Machines*

De esta forma, la condición de clasificación va a implicar que cuando aplicamos los parámetros del hiperplano w a las muestras positivas nos va tener que dar un valor mayor que más uno, mientras que cuando lo aplicamos a las muestras negativas nos va a dar un valor inferior a menos uno.

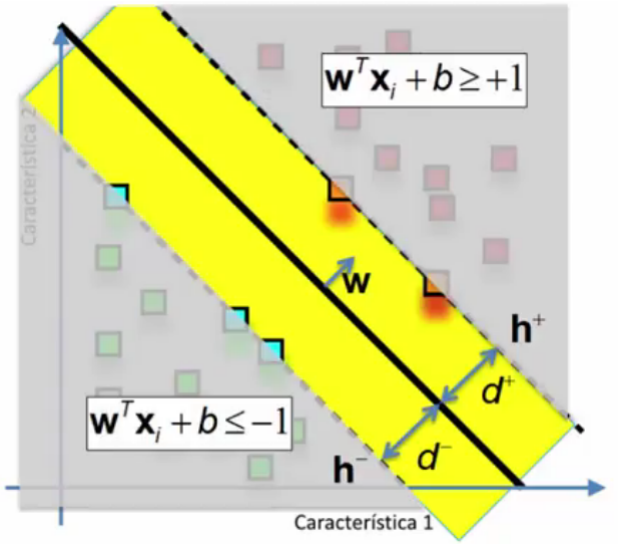

Figura 3.3: SVM: Condiciones de clasificación

Estas dos condiciones se pueden expresar conjuntamente:

$$y_i(w^T x_i + b) \geq 1 \quad (i = 1, 2)$$

De esta forma la condición de clasificación implica que todas las muestras clasificadas que están situadas en la región del espacio de características situada más allá del margen.

El margen va a corresponder a la región amarilla mostradas en las figuras 3,2 y 3,3 y se puede calcular a partir de la distancia entre los dos hiperpalanos. Si desarrollamos matemáticamente podremos observar que el margen va a depender únicamente de los parámetros w del hiperplano:

$$d^+ = d^- = \frac{|wx + b|}{||w||} = \frac{1}{||w||} \Rightarrow margen = d^+ + d^- = \frac{2}{||w||}$$

Maximizar el margen puede ser equivalente a minimizar el problema inverso:

$$Minimizar_{w,b}\phi(w) = \frac{1}{2}||w||^2$$
$$sujeto\ a:\ y_i(w^T x_i + b) \geq 1$$

Cuando a la minimización le añadimos la condición de clasificación que hemos visto antes, comprobamos que hemos transformado el problema de clasificación en un problema de optimización cuadrática estándar.

La resolución de este tipo de problemas se puede plantear con una función auxiliar que se denomina el lagrangiano:

$$L(x, \alpha) = f(x) + \sum_i \alpha_i g_i(x) \quad \forall \alpha_i \geq 0$$

siendo $f(x)$ la función a optimizar (en nuestro caso $\phi(w)$), $g_i(x)$ las restricciones ($y_i(w^T x_i + b) - 1 \geq 0$) y los factores α son los que se denominan los multiplicadores de Lagrange.
De esta forma y de manera general la solución al problema de optimización se obtiene como la minimización del lagrangiano respecto a las variables de la función a optimizar, en nuestro caso w y b, y posteriormente maximizando respecto de los multiplicadores de Lagrange.

$$\max_{\alpha}(\min_{w,b}(L(w, b, \alpha)))$$

En nuestro caso, el lagrangiano L es:

$$L(w, b, \alpha) = \frac{1}{2}w^T w - \sum_i \alpha_i[y_i(w^T x_i + b) - 1]$$

La minimización de esta expresión va a implicar realizar las derivadas parciales respecto a w y respecto a b igualando a 0:

$$\frac{\partial L}{\partial w} = w - \sum_i \alpha_i y_i x_i = 0 \Rightarrow w = \sum_i \alpha_i y_i x_i$$

$$\frac{\partial L}{\partial b} = -\sum_i \alpha_i y_i = 0 \Rightarrow w = \sum_i \alpha_i y_i = 0 \quad \alpha_i \geq 0$$

A partir de la primera derivada parcial con respecto a w vamos a obtener el valor óptimo de w como combinación lineal de las muestras de entrenamiento. Bajo condiciones de convexidad vamos a poder incluir esta solución de w en la fórmula del lagrangiano:

$$L(w,b,\alpha) = \frac{1}{2}\left(\sum_i \alpha_i y_i x_i\right)^T \left(\sum_i \alpha_i y_i x_i\right) - \sum_i \alpha_i \left[y_i (\left(\sum_j \alpha_j y_j x_j\right)^T x_i + b) - 1 \right] =$$

$$= \frac{1}{2}\left(\sum_i \alpha_i y_i x_i\right)^T \left(\sum_i \alpha_i y_i x_i\right) - \sum_i \alpha_i y_i \left(\sum_j \alpha_j y_j x_j\right)^T x_i + \sum_i \alpha_i =$$

$$= -\frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j + \sum_i \alpha_i$$

De esta forma vamos a obtener una nueva función θ cuya maximización sujeta a la condición que hemos obtenido respecto al parámetro b, nos va a dar la solución final del SVM y es un nuevo problema de optimización que es lo que se denomina SVM dual.

$$Maximizar_\alpha \theta(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j$$
$$sujeto\ a: \ \sum_i \alpha_i y_i = 0 \quad \alpha_i \geq 0$$

Así pues a partir de la formulación original que se denomina SVM primal obtenemos un nuevo problema de optimización cuya solución es la solución del SVM y que se denomina SVM dual.
En el SVM dual, podemos observar que la solución w viene dado como combinación lineal de las muestras de entrenamiento. Pero no de todas ellas ya que algunos multiplicadores de Lagrange van a ser igual a cero. Por lo tanto, solo va a ser resultado de la combinación lineal de aquellas muestras que tengan asociado un multiplicador de Lagrange mayor que cero. Estos multiplicadores de Lagrange nos van a definir qué muestras constituyen los vectores de soporte. Además, en el problema dual, las muestras aparecen como productos escalares. Este hecho lo vamos a aprovechar a continuación para poder adaptar el problema del SVM a los casos que no sean linealmente separables utilizando el concepto de *kernels*. Finalmente fijémonos que la solución de SVM que hemos obtenido de esta forma es no paramétrica, lo que quiere decir que no vamos a tener necesidad de ajustar ningún parámetro durante el entrenamiento. Sin embargo, en el caso general cuando los datos no son linealmente separables, y por tanto tenemos que introducir tolerancia a errores de clasificación, sí que va a ser necesario ajustar algunos parámetros.

A continuación vamos a ver cómo la formulación dual del problema nos ayuda a solucionar el caso en el que los datos no son linealmente separables.

Como vimos anteriormente, una estrategia consiste en definir una función de mapeo que nos convierta el espacio de características original en un nuevo espacio de características de una dimensionalidad mayor y con la esperanza de que, en este nuevo espacio de características, sí que sean linealmente separables.

- Función de mapeo:

$$x \mapsto \varphi(x)$$

- Función del Kernel:

$$K(x,z) = \varphi(x)^T \varphi(z)$$

Sin embargo, dado que en la formulación dual tenemos únicamente productos escalares entre los vectores, no va a ser necesario definir explícitamente esta función de mapeo sino que va a ser suficiente con definir un Kernel que nos defina simplemente el producto escalar en este nuevo espacio de características. Este Kernel, por lo tanto, un producto escalar se puede integrar fácilmente en la formulación dual del problema:

$$Maximizar_\alpha \theta(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

$$b = y_i - w^T \varphi(x_i) = y_i - \sum_j y_j \alpha_j K(x_j, x_i)$$

$$f(x) = sgn\left(\sum_i y_i \alpha_i K(x, x_i) + b \right)$$

Además otra de las ventajas es que no va ser necesario definir para cada problema un nuevo Kernel sino que ya tenemos diferentes tipos de Kernels estándar que en la práctica resultan bastante útiles:

- Polinomial:

$$K(x,z) = <x,z>^d$$

- Funciones de base radial:

$$K(x,z) = e^{-||x-z||^2/2\sigma} = e^{-\gamma||x-z||^2}$$

- Sigmoide:

$$K(x,z) = tanh(\kappa <x,z> - \delta)$$

- Multi-cuadrático inverso:

$$K(x,z) = (||x-z||^{1/2} 2\sigma + c^2)^{-1}$$

- Kernel de intersección:

$$K(x,z) = \sum_{i=1}^{n} min(x(i), z(i))$$

Las funciones Kernel introducen nuevos parámetros cuyo valor ha de ser fijado durante el proceso de entrenamiento, generalmente, mediante validación cruzada con algún conjunto de prueba.

La segunda alternativa para tratar el caso de datos que no sean linealmente separables consiste en suavizar la condición del margen e introducir tolerancia a errores de clasificación. La suavización del margen se va a implementar en la formulación matemática definiendo un conjunto de variables, conocidas como variables de holgura (*slack variables*), $\xi_i (\geq 0)$, que dan cuenta de aquellos vectores de soporta que violan la condición del margen.

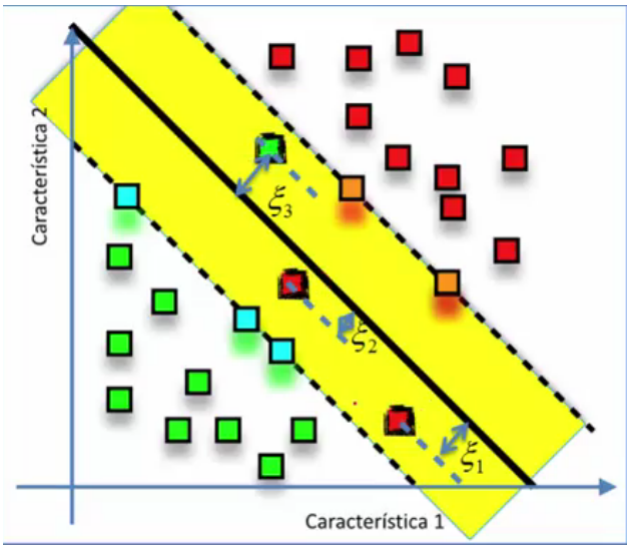

Figura 3.4: Vectores violando la condición de margen

Estas variables de holgura se incorporan en la formulación del problema relajando la condición de clasificación y cómo un nuevo factor a tener en cuenta en la función a minimizar:

$$Minimizar_{w,b,\Xi}\phi(w) = \frac{1}{2}w^T w + C\sum_i \xi_i$$
$$sujeto\ a:\ y_i(w^T x_i + b) \geq 1 - \xi_i$$

El impacto de las variables de holgura en la formulación queda modulado por el parámetro C, y este puede ser interpretado como un parámetro de regularización que nos permitirá controlar el equilibrio entre maximizar el margen y minimizar el error en las muestras de entrenamiento. Cuando este parámetro es muy grande nos va a anular la aproximación de margen suave, y va a tener que ser fijado de manera estándar con validación cruzada a partir de un conjunto de entrenamiento. En la formulación dual incluir las variables de holgura, nos va a implicar incluir una nueva inecuación que va a modular la contribución de los vectores con α distinto de cero.

$$Maximizar_\alpha \theta(\alpha) = \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j K(x_i, x_j) + \sum_i \alpha_i$$

$$sujeto\ a: \begin{cases} 0 \leq \alpha_i \leq C \\ \sum_i \alpha_i y_i = 0 \end{cases}$$

$$b = y_i(1 - \xi_i) - \sum_j y_j \alpha_j K(x_j, x_i)$$

$$f(x) = sgn\left(\sum_i y_i \alpha_i K(x, x_i) + b\right)$$

Finalmente podemos observar que en la solución final del clasificador, no influyen de manera explícita ni las variables de holgura ni el parámetro C.

4. Implementación

En esta sección, se explican los distintos métodos implementados en los diferentes ficheros que nos ayudan a resolver el problema planteado.

A partir de los ficheros que explicamos a continuación, se ha empleado `funciones.py` que contiene algunos métodos implementados en las prácticas realizadas durante el curso y empleados en este proyecto.

4.1. Descriptor HOG

En el fichero, `hog.py`, encontramos las diferentes funciones que nos ayudan a la extracción de características de una imagen determinada mediante el descriptor HOG, que se ha explicado en la sección anterior.

Como hemos explicado anteriormente se emplea el cálculo del gradiente para contruir el vector de características final. Este se calcula convolucionando la imagen con las máscaras $[1, 0, -1]$ y su transpuesta para obtener las derivadas respecto de x e y, respectivamente. Y si la imagen es a color, calculamos para cada uno de los píxeles de la imagen el gradiente para cada uno de los tres canales de color, y una vez calculado los tres canales por separado, se toma, para cada píxel, aquel canal que tiene una magnitud mayor.

Cálculo del gradiente

```
def gradiente1canale(img, border_type=cv2.BORDER_DEFAULT):
    kernel = np.matrix([-1,0,1])

    # Derivada de x
    dx = cv2.filter2D(src=1.0*img, ddepth=-1, kernel=kernel, borderType=border_type)

    # Derivada de y
    dy = cv2.filter2D(src=1.0*img, ddepth=-1, kernel=kernel.T, borderType=border_type)

    return dx, dy
```

```python
def gradienteOG(img, color=True, border_type=cv2.BORDER_DEFAULT):
    if color:
        # Obtenemos las tres matrices de colores
        b,g,r = cv2.split(img)

        # Calculamos el gradiente para cada uno de los tres canales
        dxR, dyR  = gradiente1canale(r)
        dxG, dyG  = gradiente1canale(g)
        dxB, dyB  = gradiente1canale(b)

        rows, cols = dxR.shape

        # Derivada parcial con respecto a x
        dx = np.empty((rows, cols))
        for i in range(rows):
            dx[i] = list(map((lambda j: max(dxR[i,j], dxG[i,j], dxB[i,j])), range(cols)))

        # Derivada parcial con respecto a y
        dy = np.empty((rows, cols))
        for i in range(rows):
            dy[i] = list(map((lambda j: max(dyR[i,j], dyG[i,j], dyB[i,j])), range(cols)))
    else:
        # Derivada con respecto a x e y
        dx, dy  = gradiente1canale(img)

    # Orientación del gradiente (en grados)
    orientamento = np.array( (np.arctan2(dy, dx)*180/np.pi)

    # Magnitud del gradiente
    grandezza = np.sqrt(dx**2 + dy**2)

    return orientamento, grandezza
```

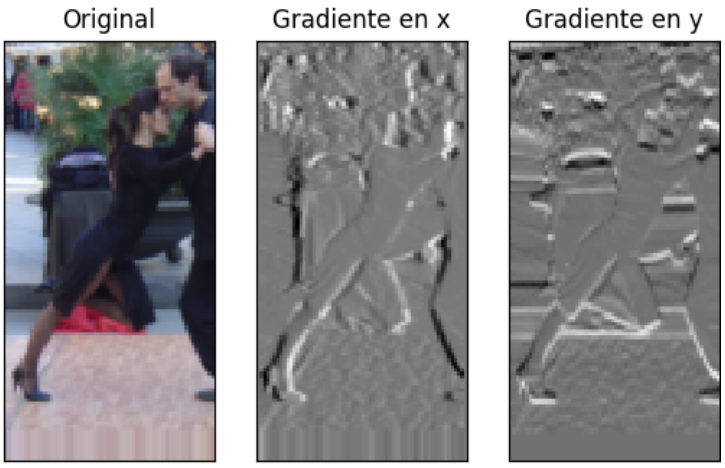

Figura 4.1: Gradientes.

Figura 4.2: Magnitud del gradiente.

A continuación se implenta HOG siguiendo exactamente los mismos pasos que se ha descrito en la sección anterior y ayudándonos de las funciones auxiliares `pesok`, `pesox` y `pesoy` para las distintas ponderaciones que antes se ha explicado anteriormente, `istogramma` para la contrucción del histograma de una determinada celda (para evitar cálculos innecesarios, consideramos 1/4 de los bloques más cercanos al píxel que estemos evaluando, ya que fuera de esa región valdrá cero), `normalizzazione` para la normalización de un determinado vector, y por último, la función `hog` que calcula dicho descriptor con un tamaño de celda, tamaño del bloque y número de intervalos del histograma pasado por parámetros, y considerando la orientación gradiente sin signo y un solapamiento de blo-

que de 3/4.

Histograma de una celda

```
# Factor de asignación de cada gradiente a uno de los intervalos del histograma
# max(0, 1-(angolo - (k*grado + (k-1)*grado)/2)/ grado )
# = max(0, 1-(angolo - (2*k-1)*grado/2)/ grado )
# = max(0,1-(angolo/grado - (2*k-1)/2 ))
# = max(0,1-( angolo/grado - k+1/2 ))
def pesok(angolo, k, grado=20):
    if angolo>180:
        angolo = -(360-angolo)

    return max(0,1-abs(angolo/grado-k+0.5))

# Factor de asignación de un píxel a una celda en la dirección x e y
pesox = lambda centerx, abscissa, dx: max(0, 1-abs(abscissa-centerx)/dx)
pesoy = lambda centery, ordinate, dy: max(0, 1-abs(ordinate-centery)/dy)

# Histograma de la celda ij
#   img = imagen
#   cell = celda (i,j)
#   theta = orientaciones del gradiente
#   g = magnitudes del gradiente
#   cell_size = tamaño de la celda
#   bins = número de intervalos del histograma
def istogramma(img, theta, g, cell, cell_size=(8,8), bins=9, grado=20):
    # Centro de la celda cell
    center = (cell_size[0]*((2*cell[0]+1)/2), cell_size[1]*((2*cell[1]+1)/2))

    # Tenemos en cuenta sólo las celdas de alrededor
    num_cells = (img.shape[0]//cell_size[0], img.shape[1]//cell_size[1])
    cell_inizio = (max(0, cell[0]-0.5), max(0,cell[1]-0.5))
    cell_fine = (min(num_cells[0]-1, cell[0]+0.5), min(num_cells[1]-1, cell[1]+0.5))

    inizioI = (cell_inizio[0]*cell_size[0], cell_inizio[1]*cell_size[1])
    fineI = ((cell_fine[0]+1)*cell_size[0], (cell_fine[1]+1)*cell_size[1])

    # Construimos el histograma
    h = np.zeros(bins)
    for k in range(1, bins+1):
        for i in range(int(inizioI[0]), int(fineI[0])):
            for j in range(int(inizioI[1]), int(fineI[1])):
                magnitude = g[i,j]
                if magnitude!=0:
```

```
                    weightx = pesox(center[1], j, cell_size[1])
                    if weightx!=0:
                        weighty = pesoy(center[0], i, cell_size[0])
                        if weighty!=0:
                            h[k-1] +=  weightx * weighty \
                                        * pesok(theta[i,j], k, grado) * magnitude

    return h
```

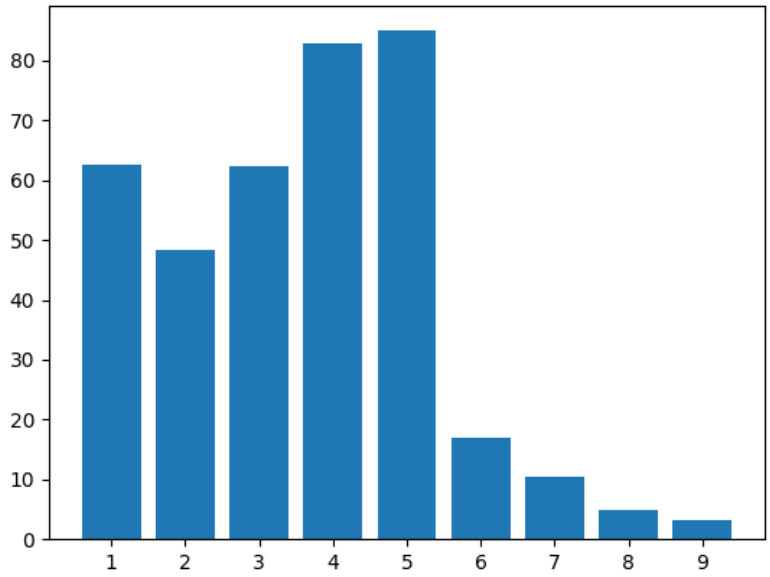

Figura 4.3: Histograma de una de las celdas de la fotografía anterior.

Descriptor HOG

```
# Normalización del vector bloque v:
    # v' = v+sqrt(||v||^2 + eps) siendo ||.|| la
    # norma euclídea y eps una constante con un valor muy pequeño
    # para evitar divisiones por cero.
normalizzazione = lambda v, eps: v/np.sqrt(v.dot(v)+eps)

# Utilizaremos el gradiente sin signo
#   cell_size = tamaño de las celdas
#   block_size = tamaño del bloque
#   bins = número de intervalos del histograma
def hog(filename, cell_size=(8,8), block_size=(2,2), bins=9, eps=10**(-25), read=True):
    if read:
        # Leemos la imagen
        img = cv2.imread(filename)
```

```
else:
    img = filename

# Calculamos la orientación y la magnitud de la imagen
orientation, magnitude = gradienteOG(img)

# Dividimos la imágen en un número fijo de celdas y para cada una
# de estas se obtiene un histograma de las orientaciones de los
# gradientes de esa celda.
# Se calculan los histogramas para todas las celdas
grado = 180//bins
numCells = (img.shape[0]//cell_size[0], img.shape[1]//cell_size[1])

histograms = np.ndarray(numCells, dtype='O')
for i in range(numCells[0]):
    for j in range(numCells[1]):
        histograms[i,j] = istogramma(img, orientation, magnitude, (i,j),
                                     cell_size, bins, grado)

# Todos los histogramas anteriores se combinan para obtener la
# representación global de toda la imagen en forma de vector de
# características

        # Para cada bloque vamos a coger los histogramas de cada una
        # de las celdas y concatenarlos para obtener el vector con la
        # representación del bloque. La normalización de los histogramas
        # va a ser realizada a nivel de bloque, es decir, vamos a
        # normalizar este bloque resultado de concatenar todas las celdas.

# Solapamiento de bloque de 3/4
numBlocks = (numCells[0]-block_size[0]+1, numCells[1]-block_size[1]+1)

# Número de celdas/bloque
numCB = block_size[0]*block_size[1]

# Vector solución
lim = numCB*bins
hog = np.ndarray(numBlocks[0]*numBlocks[1]*lim, dtype='O')

# Histograma de gradientes
s = 0
for i in range(numBlocks[0]):
    row = min(numBlocks[0]+1, i+block_size[0])
    for j in range(numBlocks[1]):
```

```
            col = min(numBlocks[1]+1, j+block_size[1])

            # Concatenemos los histogramas del mismo bloque
            v = np.concatenate(np.concatenate(histograms[i:row, j:col].tolist()))

            # Agregamos los nuevos histogramas normalizados al vector solución
            hog[s:s+lim] =  normalizzazione(v, eps)
            s += lim

    return hog
```

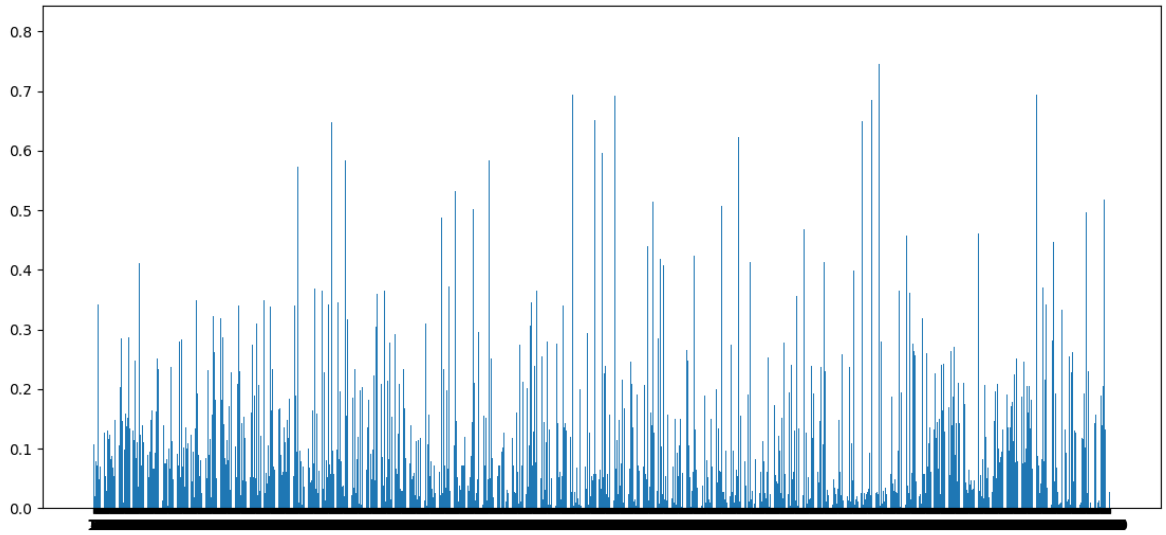

Figura 4.4: Vector de características de la fotografía anterior.

4.2. Extracción de características

En el fichero `extraer_caracteristicas.py` se pretende extraer todas las características de la carpeta `Train` donde se encuentra dos subcarpetas denominadas `pos`, donde aparecen las distintas imágenes que contienen peatones, y `neg`, donde aparecen imágenes con fondo (entendiendo por fondo, todo aquello que no es peatón). Los distintos vectores de características se almacenan en los ficheros que `<nombre_carpeta>.txt` que guardaremos en la carpeta `Annotations`.

Extracción de características

```
def main(argv,):
    ruta = './Train/'
```

```
# Nombres de todas las imágenes que contienen peatones
nomi_pos = os.listdir(ruta+'pos')

# Nombres de todas las imágenes que contienen fondo u otros objetos
nomi_neg = os.listdir(ruta+'neg')

# Abrimos los distintos archivos para guardar las características
# de las distintas imágenes
pos = open("./Annotations/pos.txt", "w")
i = 1
for nome in nomi_pos:#[:1000]:
    print(i, " Extrayendo caracteristicas de...", nome)
    h = hog(ruta+"pos/"+nome)
    c = ' '.join(map(str, h))
    pos.write(c+"\n")
    i+=1
pos.close()

print()
print("------------------------------------------------------------")
print()

neg = open("./Annotations/neg.txt", "w")
i = 1
for nome in nomi_neg:
    print(i, "Extrayendo caracteristicas de...", nome)
    h = hog(ruta+"neg/"+nome)
    c = ' '.join(map(str, h))
    neg.write(c+"\n")
    i+=1
neg.close()

pass
```

NOTA: Si se desea ejecutar dicho programa, se recomienda no hacerlo con todas las imágenes de la carpeta ubicadas actualmente en `Train` ya que el algoritmo tarda aproximadamente 2 segundos para cada imagen de 64×128 píxeles. Por tanto, muchas imágenes implica mucho tiempo de ejecución, y si se interrumpe en mitad se pueden perder todos los datos de los ficheros dedicados a este propósito.

4.3. Entrenamiento

En el fichero `entrenar.py` nos encontramos las siguientes funciones

- `leggere`: nos permite cargar todos los vectores de características del fichero pasado como argumento.

Carga de los vectores de características

```
def leggere(filename):
    file = open(filename, "r")

    # Leemos cada línea del fichero (vector de característica de cada imagen)
    # y la transformamos en un vector de reales
    linee = file.readlines()
    matrice = []
    for linea in linee:
        matrice.append( np.array([[m.strip() for m in n]
                                  for n in [linea.split(" ")]][0],
                                  dtype=float) )
        # for linea in linee]][0],


    # Convertimos la lista en una matriz
    N = len(matrice) # Número de muestras
    M = len(matrice[0]) # Tamaño del vector de características
    matrice = np.array(matrice).reshape(N, M)

    return N, matrice
```

- `allenare`: Entrenamos las muestras a partir del clasificador SVM, pudiendo seleccionar si deseamos emplear el SVM lineal, mediante el parámetro `lineal=True`, o SVM con kernel gaussiano y ancho de kernel $3e-2$, además se puede especificar parámetro de penalización `C` del término de error.

Entrenamiento con SVM

```
# Entramiento con SVM. Guardamos el modelo resultante
def allenare(matricePositiva, matriceNegativa, lineal=True, C=0.01):
    Npositive = len(matricePositiva) # Número de muestras positivas
    Nnegative = len(matriceNegativa) # Número de muestras negativas

    # Inicializamos la estructura que pasaremos al modelo para entrenar
    # X -> muestras a entrenar
    # y -> etiquetas (+1 si en la imagen se encuentra un peatón
    #                 y -1 en caso contrario )
    X = np.concatenate((matricePositiva, matriceNegativa), axis=0)
    y = np.append(np.ones(shape=(1, Npositive)),
                  -1*np.ones(shape=(1, Nnegative)))

    if lineal: # SVM lineal
    # dual : bool, (default=True)->Select the algorithm to either solve the dual
```

```
    # or primal optimization problem. Prefer dual=False when n_samples > n_features.
        M = len(matricePositiva[0]) # Tamaño del vector de características
        if (Npositive+Nnegative)>M:
            dual = False
        else:
            dual = True
        model = LinearSVC(C = C, dual = dual)
    else: # SVM con kernel gaussiano
        model = SVR(kernel = 'rbf', C = C, gamma = 3e-2)

    model.fit(X, y)

    # Porcentaje de acierto para el conjunto de entrenamiento
    successo(matricePositiva, matriceNegativa, model)

    # Guardamos el modelo
    modello = open("./Annotations/model.txt", "wb")
    pickle.dump(model, modello)

    return model
```

- `successo`: nos muestra el porcentaje de acierto totales para una determinada muestra.

```
def successo(matricePositiva, matriceNegativa, model):
    erroriPos = 0
    erroriNeg = 0

    Npositive = len(matricePositiva) # Número de muestras positivas
    Nnegative = len(matriceNegativa) # Número de muestras negativas

    for vettore in matricePositiva:
        if model.predict([vettore])<=0:
            erroriPos += 1

    for vettore in matriceNegativa:
        if model.predict([vettore])>=0:
            erroriNeg += 1

    # Porcentaje de acierto para el conjunto dado
    print("Porcentaje de acierto:", 1-(erroriPos+erroriNeg)/(Npositive + Nnegative))
```

Con la ayuda de todas estas funciones se pretende mostrar el porcentaje de aciertos del conjunto de entrenamiento (que por defecto se hará con el 70 % de las imágenes de

las cuales se conoce su vector de características), y del conjunto de prueba (imágenes restantes).

Porcentaje de aciertos

```
def main(argv,):
    ## Cargamos las características extraídas de las distintas imágenes ##
    # Imágenes con peatones
        # Npositive: Número de muestras positivas
        # M: Tamaño del vector de características
        # matricePositiva: vector de características de cada muestra con peatones
    Npositive, matricePositiva = leggere("./Annotations/pos.txt")

    # Imágenes con fondo y otros objetos
        # Nnegative: Número de muestras negaticas
        # M: Tamaño del vector de características
        # matriceNegativa: vector de características de cada muestra sin peatones
    Nnegative, matriceNegativa = leggere("./Annotations/neg.txt")


    ## ENTRENAMIENTO ##
    # Mezclamos las matrices de forma aleatoria (Las matrices multidimensionales
    # sólo se mezclan a lo largo del primer eje)
    np.random.shuffle(matricePositiva)
    np.random.shuffle(matriceNegativa)
    # Entrenamos SVM con el (percentuale*100)% de muestras positivas y de negativas
    percentuale = 0.7
    S = int(percentuale*Npositive)
    T = int(percentuale*Nnegative)
    model = allenare(matricePositiva[:S], matriceNegativa[:T])

    ## TEST ##
    if S!=Npositive:
        successo(matricePositiva[S:Npositive], matriceNegativa[T:Nnegative], model)

    pass
```

4.4. Test imagen

Por último, en el fichero `text_imagen.py` se implementa la función encargada de encontrar un peatón en un imagen de tamaño arbitrario. Para ello, se crea la pirámide Gaussiana de `n` niveles y, para cada nivel, se emplea el método de ventana deslizante.

La ventana deslizante es una región rectangular de anchura y altura fija, en este caso de 64×128 por defecto, que se "desliza" sobre el eje horizontal y vertical de la imagen.

Para cada una de estas ventanas, extraemos el vector de características y aplicamos el clasificador SVM para determinar si hay persona o no. El clasificador nos devuelve un valor que indica cuanto se acerca esa ventana a una clase o a otra. Si ese valor supera cierto umbral predefinido, podemos considerar que ahí hay un peatón. La siguiente posición de la ventana se calcula sumándoe un valor fijo de píxeles a la anterior en el eje correspondiente, siendo recomendable que se superpongan.

Detección de peatones

```
# Detectar peatón en la imágen si lo hubiese
    # filename: nombre de la imágen
    # finestra: tamaño de la ventana
    # salto: número de píxeles que vamos a saltar para deslizar la ventana
    #por toda la imágen
    # n: niveles de la pirámide Gaussiana
    # soglia = umbral para determinar si la predicción es correcta
    # level: primer nivel a considerar
    # lineal: True si hemos SVM lineal y False en caso contrario
def scoprirePedone(filename, finestra=(128, 64), salto=(64, 32), n=3, soglia=0,
                   level=0, lineal=True):
    # Cargamos el modelo entrenado
    with open("./Annotations/model.txt","rb") as fd:
        model=pickle.load(fd)

    # Pirámide Gaussiana de n niveles
    piramide = pyrGauss(filename, levels=n)
    imagen = piramide[0]

    for scala, img in enumerate(piramide[level:]):
        (N, M, s) = img.shape
#         im = img.copy()
        for i in range(0, N, salto[0]):
            for j in range(0, M, salto[1]):
                if (i+finestra[0])>N and (j+finestra[1])<=M \
                        and (i+finestra[0]-N)<5:
                    res = np.empty((finestra[0], finestra[1], 3) )
                    n = N-i
                    w = 0#differenza//2
                    res[w:w+n] = img[i:i+finestra[0], j:j+finestra[1]]
                    res[w+N:] = img[-1, j:j+finestra[1]]
                else:
                    res = img[i:i+finestra[0], j:j+finestra[1]]

                if res.shape[0]==finestra[0] and res.shape[1]==finestra[1]:
#                     cv2.rectangle(im,(j,i), (j+finestra[1], i+finestra[0]),
```

```
#                                               (255,0,0), 2)
#                     visualization([im], [''], 1, 1, color=[True])

                    # Hallamos el vector de características de la subimagen
                    vector = hog(res, read=False)

                    # Obtenemos la predicción de las puntaciones de confianza
                    # para las muestras
                    if lineal:
                        decision_func = model.decision_function([vector])
                    else:
                        decision_func = model.predict([vector])

                    # Si es mayor un umbral prefijado, la consideramos
                    if decision_func>=soglia:
#                        print(decision_func)
                        s = 2**(scala+level)
                        cv2.rectangle(imagen,(s*j,s*i), (s*(j+finestra[1]),
                                                         s*(i+finestra[0])),
                                                         (0,255,0), 2)

    visualization([imagen], [''], 1, 1, color=[True])
```

Nota: Por comidad, si desea ejecutar este método con una imagen determinada insértela en la carpeta `images` con el nombre `prueba.png`. Si desea cambiar este nombre por razones de gusto o extensión de la imagen, se deberá modificar el nombre en `main` asociado a este fichero.

4.5. Generar muestras

En el fichero `crear_muestras.py` se ha creado varias funciones que nos permiten generar nuevas muestras para entrenar y probar nuestro modelo. Aunque en este proyecto al final no se ha usado ya que hemos encontrado una base de datos que nos los proporcionaban. En primer lugar está la función `redimensionare` que nos permite redimensionar la fotografía manteniendo las proporciones.

Redimensionar

```
def ridimensionare(filename, dim=(128, 64)):
    img  = cv2.imread(filename)
    n = img.shape[0] # Alto
    m = img.shape[1] # Ancho

    sobrepasa = False
```

```
    width = dim[1]
    percent = width/float(m)

    if n*percent>dim[0]:
        sobrepasa = True
        img = img.transpose((1,0,2))
        dim = (dim[1], dim[0])
        height = dim[1]
        percent = height/float(n)

    # Redimensionamos la foto manteniendo las proporciones
    img = cv2.resize(img, None, fx=percent, fy=percent, interpolation = cv2.INTER_CUBIC)

    # Rellenamos para que tenga las dimensiones deseadas
    res = np.empty((dim[0], dim[1], 3) )
    differenza = dim[0] - img.shape[0]
    i = differenza//2
    height = img.shape[0]

    res[0:i] = img[0]
    res[i:i+height] = img[:height]
    res[i+height:] = img[height-1]

    # Volvemos a ponerla bien
    if sobrepasa:
        res = res.transpose((1,0,2))


    # Guardamos la imagen
    cv2.imwrite(filename, res)
```

Mientras que la función **??** nos permite, como su nombre indica, recortar la subimagen central definida por el rectángulo $(xmin, ymin)$ y $(xmax, ymax)$ que son el vertice superior a la izquierda y el inferior a la derecha con un margen de 8 píxeles.

Recortado de imágenes

```
def recortar(filename, xmin, ymin, xmax, ymax, num):
    img  = cv2.imread(filename)

    filename = filename.lower()
    nome = filename[:len(filename)-4]+"_"+str(num)+filename[len(filename)-4:]
    cv2.imwrite(nome, img[ymin-8:ymax+8, xmin-8:xmax+8])

    return nome
```

Ayudándonos de las funciones anteriores se leía las muestras de la base de datos [7] y se recortaban las imágenes al tamaño especificados en los ficheros.

Crear muestras

```
def leggereNumero(linea, inizio):
    numero = linea[inizio]

    if len(linea)>inizio:
        fin = inizio+1
        letra = linea[fin]

    while letra!=',' and letra!=')' and len(linea)>fin:
        numero += letra
        fin += 1
        letra = linea[fin]

    return int(numero), fin

def main(argv,):
    fileI = open('Train/pos.lst')
    imgs = []
    for immagine in fileI.readlines():
        print("Leyendo...", immagine)
        im = str("./"+immagine)
        im = im[:len(im)-1]
        imgs.append(im)

#         ridimensionare(im)


    # Recortamos por donde se encuentra la persona
    file = open('Train/annotations.lst')
    numI = 0
    for annotazione in file:
        nota = annotazione[:len(annotazione)-1]
        num = 1
        fileA = open(nota)
        print("->Leyendo...", nota)
        for linea in fileA.readlines():
            testo =  'Bounding box for object '+str(num)+' "PASperson" (Xmin, Ymin) - (Xma
            inizio = len(testo)
            xmin, inizio = leggereNumero(linea, inizio)
            ymin, inizio = leggereNumero(linea, inizio+2)
```

```
            xmax, inizio = leggereNumero(linea, inizio+5)
            ymax, inizio = leggereNumero(linea, inizio+2)

            nome = recortar(imgs[numI], xmin, ymin, xmax, ymax, num)
            ridimensionare(nome, dim=(160, 96))
            num+=1
        numI+=1
    pass
```

5. Resultados experimentales

Debido a problemas con la base de datos proporcionada, se ha utilizado otra relacionada, que se encuentra en la consigna de la UGR [8] extraida de [1]. Además, si se quiere probar el modelo para una determinada imagen sin ejecutar todo el proceso, también se puede obtener el fichero con los vectores de características y el modelo entrenado con SVM lineal en [9].

En esta sección, se expondrán los resultados obtenidos tras varias ejecuciones entrenando el 70 % de datos extraidos aleatoriamente de la base de datos contenida en la carpeta `Train` mediante SVM lineal o con kernel Gaussiano. Y posteriormente se muestra aplicado a una imagen determinada de un tamaño arbitrario empleando el método de ventana deslizante.

5.1. Resultados de las distintas ejecuciones

A continuación se muestra dos tablas que contienen los siguientes porcentajes, diferenciando los datos del entrenamiento (*train*) del *test*:

- TPR(*True Positive Rate*): Proporción de verdaderos positivos bien clasificados. Es decir, esta sería la probabilidad de detección de un peatón calculada como $1 - \frac{FN}{N}$ siendo FN el número de muestras clasificadas como negativas cuando si que contenían peatones y N el número total de muestras.
- TNR(*True Negative Rate*): Proporción de verdaderos negativos bien clasificados calculada como $1 - \frac{FP}{N}$ siendo FP el número de muestras clasificadas como positivas cuando no contenían ninguna persona. Por tanto, la probabilidad de falsa alarma es 1-TNR= $\frac{FP}{N}$.
- TR: Probabilidad de acierto ($1 - \frac{FN+FP}{N}$).

Ejecuciones realizadas entrenando las muestras con SVM lineal y $C = 0{,}01$

	TPR	TNR	TR
Train 1	0.9894291754756871	0.9947145877378436	0.9841437632135307
Train 2	0.9878435517970402	0.99392177589852	0.9817653276955602
Train 3	0.9886363636363636	0.9957716701902748	0.9844080338266384
Train 4	0.9894291754756871	0.9933932346723044	0.9828224101479915
Train 5	0.9904862579281184	0.9949788583509513	0.9854651162790697
Train (media)	0.989164905	0.994556025	0.98372093
Test 1	0.9716399506781751	0.9778051787916153	0.9494451294697904
Test 2	0.9817653276955602	0.9827373612823674	0.9568434032059187
Test 3	0.969173859432799	0.9778051787916153	0.9469790382244143
Test 4	0.9753390875462392	0.9778051787916153	0.9531442663378545
Test 5	0.9753390875462392	0.9827373612823674	0.9580764488286067
Test (media)	0.974651463	0.979778052	0,952897657

Tabla 5.1: SVM lineal ($C = 0{,}01$).

Ejecuciones realizadas entrenando las muestras con SVM kernel Gaussiano con $gamma = 3e-2$ y $C = 5$

	TPR	TNR	TR
Train 1	1.0	1.0	1.0
Train 2	1.0	1.0	1.0
Train 3	1.0	1.0	1.0
Train 4	1.0	1.0	1.0
Train 5	1.0	1.0	1.0
Train (media)	1.0	1.0	1.0
Test 1	0.9796547472256474	0.9882860665844636	0.967940813810111
Test 2	0.9734895191122072	0.9876695437731196	0.9611590628853267
Test 3	0.9815043156596794	0.9864364981504316	0.967940813810111
Test 4	0.9821208384710234	0.9882860665844636	0.9704069050554871
Test 5	0.9827373612823674	0.9858199753390875	0.968557336621455
Test	0.979901356	0,98729963	0,967200986

Tabla 5.2: SVM kernel Gaussiano ($gamma = 3e-2$, $C = 5$).

No podemos concluir que SVM con kernel Gaussiano nos vaya a proporcionar mejores resultados ya que no hay una diferencia demasiado apreciable con respecto a este conjunto de datos. Dependiendo de la imagen, nos proporcionará uno mejores (o iguales) resultados con uno o con otro. Además con kernel Gaussiano tarda mucho en entrenar las muestras produciendo un sobreajuste a pesar de permitirle un cierto error. Veamos a continuación cómo se comportan estos modelos en imágenes de mayor tamaño.

5.2. Aplicación a imágenes

En esta sección se muestran una serie de ejemplos en los que se ha usado ventana deslizante de 64×128 (cada 32 y 64 píxeles horizontal y verticalmente, respectivamente) para la detección de peatones (si los hubiese), entrenando las muestras con SVM tanto lineal como el kernel Gaussiano con los parámetros antes expuestos.

Se ha establecido un umbral de predicción bajo ($soglia = 0$) para poder mostrar algunos de los problemas que se producirían en otras imágenes aun poniendo el umbral con una restricción mayor.

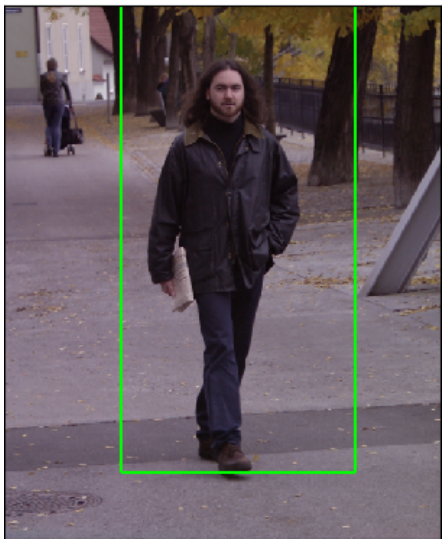

Figura 5.1: Detección de peatón de forma correcta.

Figura 5.2: Fondo.

Figura 5.3: Detección de peatón de forma correcta con SVM lineal.

Figura 5.4: Detección de peatón de forma incorrecta con SVM kernel Gaussiano.

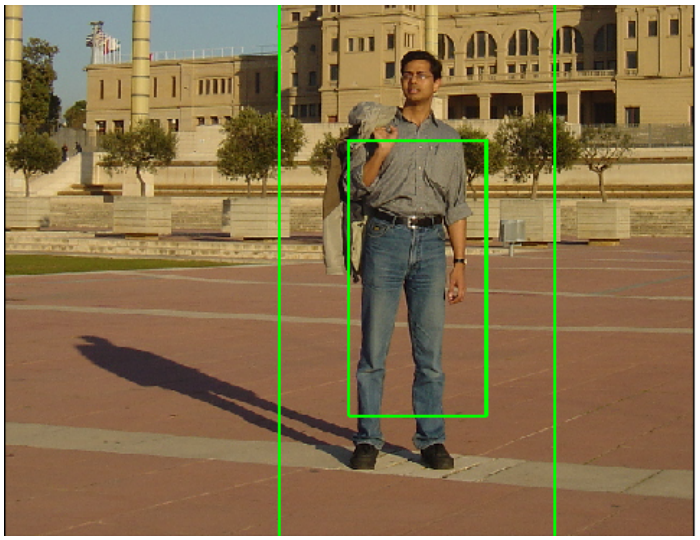

Figura 5.5: Detección de peatón de forma correcta e incorrecta.

Figura 5.6: No se detecta al peatón.

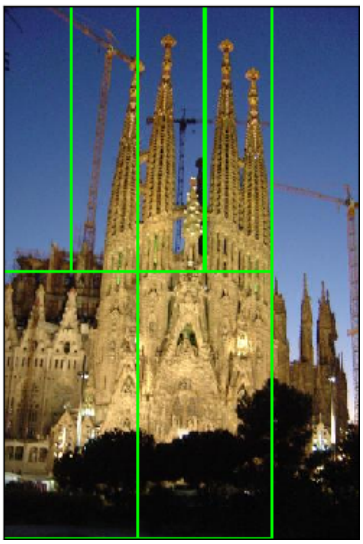

Figura 5.7: Detección de peatón de forma incorrecta con SVM lineal.

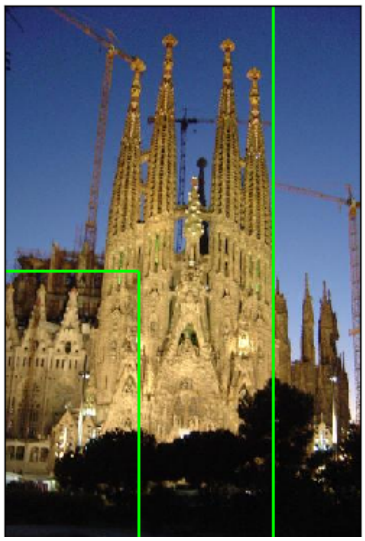

Figura 5.8: Detección de peatón de forma incorrecta con SVM kernel Gaussiano.

Podemos observar que en algunos casos detectan peatones (si los hay) correctamente ambos métodos, como se muestra en 5.1, 5.3 y 5.2. Pero pueden producirse casos en que detecten sólo una parte de este, como ocurre en las figuras 5.4 o 5.1, en alguno de los niveles de la pirámide Gaussiana, o directamente no lo detecte como sucede en la figura 5.6 (en este caso, por no posicionarse ninguna ventana de forma correcta). Por el contrario, también pueden producirse falsos positivos como ocurre notablemente en 5.7 y 5.8.

Muchos de estas detecciones incorrectas se solucionarían aumentando el umbral de predicción ya que la mayoría de los errores de las distintas imágenes son debidos a que los valores de predicción obtenidos han sido muy cercanos a cero, pero, en consecuencia, el peatón de la imagen de la figura 5.3 no se detectaría con ninguno de los modelos entrenados. Por tanto, la elección de este umbral es un importante parámetro a determinar.

6. Propuestas de mejora

Como primera consideración, deberíamos de hacer validación cruzada para determinar los valores mejores de parámetros tan importante como, por ejemplo, C ayudándonos de la curva ROC para encontrar un equilibrio entre la proporción de verdaderos positivos y verdaderos negativos, o del ancho de kernel (si se desea usar SVM con kernel Gaussiano), etc.

Como se explica en [6], se podría realizar un preprocesamiento mediante la normalización de Color/Gamma. Y sería recomendable emplear *Non-Maxima-Suppression (NMS)* para suprimir los rectángulos que se solapan con cierto margen, además de elegir un factor de escala para construir los distintos niveles de la pirámide ya que eso nos permitirá seleccionar el rectángulo más ajustado al área donde se encuentra el peatón, a

parte de elegir un umbral de predicción más certero, como ya hemos comentado antes. Además debería producirse mayor solapamiento entre las distintas subimágenes, del método de ventana deslizante, para que sea más probable detectar del peatón (si lo hubiese).

Por otro lado, la división de la imagen en bloques de HOG es muy similar a la que se realiza en el descriptor LBP. Esta coincidencia en la división en bloques, favorece que con la misma división de la imagen en bloques se puedan fácilmente calcular ambos descriptores, LBP y HOG. Esta facilidad permite diseñar métodos de detección que calculen los dos descriptores a la vez y posteriormente los combine. De esta forma habitualmente se mejoran los resultados de detección, ya que combinamos las propiedades de textura y de contorno que nos proporcionan ambos descriptores.

7. Conclusión

Por todo lo visto, es claro que se debe hacer un análisis profundo antes de abordar cualquier problema y además realizar muchas pruebas de forma práctica para poder determinar los parámetros de los distintos métodos (si los necesitase) de la manera más óptima posible. Sin embargo, en este proyecto hemos podido acercarnos a la solución del problema a partir de técnicas empleadas por la visión por computador que han dado buenos resultados en distintas áreas y, además, hemos podido observar la importancia del gradiente cuando trabajamos con imágenes.

Referencias

[1] https://www.coursera.org/learn/deteccion-objetos/

[2] https://academica-e.unavarra.es/bitstream/handle/2454/24591/Memoria.pdf?sequence=1&isAllowed=y

[3] http://www.ia.uned.es/~ejcarmona/publicaciones/%5B2013-Carmona%5D%20SVM.pdf

[4] http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[5] http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

[6] http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf

[7] http://pascal.inrialpes.fr/data/human/

[8] https://consigna.ugr.es/f/AMYEpWsI9e6YrN8t/Datasets.zip

[9] https://consigna.ugr.es/f/J69qMLyhM4qRiXAx/Annotations.zip