ESCUELA TECNICA DE INGENIERIAS INFORMATICA Y
TELECOMUNICACIONES

Actividades sobre descriptores:

Extracción de Características en Imágenes

Autor:
Nicolás Delgado Guerrero

Índice

1. Mejoras Obligatorias.

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

En esta sección del presente trabajo explicaremos como hemos entrenado un clasificador SVM para poder predecir si en una fotografía hay patones o no. Para ello cargaremos los datos sobre el conjunto de imágenes dadas por el profesor. En este conjunto tenemos 5406 imágenes, de las cuales usaremos 4306 para entrenamiento (1916 de peatones y 2390 de fondo) y 1100 imágenes para test (500 de peatones y 600 de fondo).

El proceso para cargar las imágenes lo haremos sobre la función cargar_datos() definida en el script cargar_imagenes_datos.py. Esta función retorna dos Numpy Arrays, uno con un descriptor de las imágenes y otro con las clases asociada a esa imagen. El descriptor que usaremos para las imágenes es el HoG. HoG significa Histogramas de Gradientes Orientados, este es un tipo de "descriptor de características". La idea esencial detrás del descriptor es la información del gradiente en cada píxel, con ello podemos averiguar como cambia el píxel en las diferentes direcciones.

Definimos las siguientes variables.

```
PATH_POSITIVE_TRAIN = "data/train/pedestrians/"
PATH_NEGATIVE_TRAIN = "data/train/background/"
PATH_POSITIVE_TEST = "data/test/pedestrians/"
PATH_NEGATIVE_TEST = "data/test/background/"
IMAGE_EXTENSION = ".png"
```

En ellas se encuentran las rutas de las imágenes de entrenamiento y test de peatones y fondo respectivamente. Usaremos la función cargar_datos() de cargar_imagenes_datos.py para construir los conjuntos de entrenamiento y test con sus respectivas clases.

```
training_data, classes_train = cargar_imagenes_datos.cargar_datos(PATH_POSITIVE_TRAIN,PATH_NEGATIVE_TRAIN)
test_data, classes_test = cargar_imagenes_datos.cargar_datos(PATH_POSITIVE_TEST,PATH_NEGATIVE_TEST)
```

En la función cargar_datos() leemos imagen por imagen y a esta le calculamos su descriptor HoG haciendo uso de la biblioteca OpenCv. Vamos concatenando los descriptores de cada imagen en una lista y en otra lista le asignamos la clase de esa imagen. Primero se hace para las imágenes de peatones.

```
for filename in os.listdir(PATH_POSITIVE):
    if filename.endswith(IMAGE_EXTENSION):
        filename = PATH_POSITIVE+filename
        img = cv2.imread(filename)
        hog = cv2.HOGDescriptor()
        descriptor = hog.compute(img)
        data.append(descriptor)
        clases.append(1)
        counter_positive_samples += 1
        print("Leidas " + str(counter_positive_samples) + " imágenes de -> peatones")
```

De forma análoga calculamos los descriptores y las clases para las imágenes de fondos y se concatenan en las listas data y classes. Finalmente la función retorna estas dos listas transformandolas previamente en variables de tipo Numpy Arrays.

```
return np.array(data), np.array(clases)
```

Por último hacemos un merge a los conjuntos de entrenamiento y test. A este nuevo conjunto le haremos una extracción de 5 subconjuntos de forma estratificada sobre los cuales realizaremos una validación cruzada del clasificador. Para formar los subconjuntos o folds de forma estratificada usaremos la función Kfolds() de la biblioteca skalearn. Finalmente obtendremos 5 conjuntos de datos de entrenamiento y test para los descriptores de las imágenes y lo mismo para las clases.

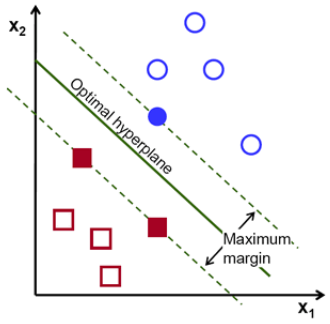
```
data = np.concatenate((training_data, test_data), axis = 0)
classes = np.concatenate((classes_train, classes_test), axis = 0)

kf = KFold(n_splits = 5, shuffle = True)

train_data = []
train_classes = []
test_data = []
test_classes = []

for tr, tst in kf.split(data, classes):
    train_data.append(data[tr, :])
    train_classes.append(classes[tr])
    test_data.append(data[tst, :])
    test_classes.append(classes[tst])
```

Obtenemos listas de cinco elementos donde cada elemento es un subconjunto ya sean de los datos de train, test o las clases de train y test. Ahora para cada uno uno de los índices de las listas entrenaremos un clasificador y calcularemos sus predicciones, además usaremos distintas métricas de bondad para la clasificación.

```
for i in range(0, 5):

    svm_modelo = clasificador_SVM.entrenar_clasificador(train_data[i], train_classes[i], "LINEAR")
    print("Clasificador entrenado")

    prediccion = clasificador_SVM.predecir_clasificador(test_data[i], svm_modelo)

    bondad_metricas = []

    bondad_metricas.append(metrics.accuracy_score(test_classes[i], prediccion))
    print("Accuracy: "+str(bondad_metricas[0]))

    bondad_metricas.append(metrics.precision_score(test_classes[i], prediccion))
    print("Precision score: "+str(bondad_metricas[1]))

    bondad_metricas.append(metrics.f1_score(test_classes[i], prediccion))
    print("F_1 score: "+str(bondad_metricas[2]))

    bondad_metricas.append(metrics.recall_score(test_classes[i], prediccion))
    print("Recall: "+str(bondad_metricas[3])+"\n")
```

Para entrenar el clasificador usaremos la función entrenar_clasificador() y para predecir lo haremos mediante predecir_clasificador(), ambas funciones están definidas en el script clasificador_SVM.py y veremos más detalladamente en adelante.

En este caso el kernel que vamos a usar para el SVM es el lineal, a groso modo podríamos decir que el espacio donde se encuentran los datos va a ser separado por un hiperplano; en dos dimensiones ese hiperplano sería una recta.



Las predicciones van a ser comparadas con los valores reales de las clases que teníamos almacenadas en nuestra lista test_classes. Si hacemos una tabla de doble entrada donde las columnas representan las predicciones y las filas las etiquetas reales de la clase podemos calcular cuanto hemos acertado, el número de fallos, etc. Esta tabla se denomina matriz de confusión y a partir de ella se puede conocer bastante bien como de buenas son las predicciones de nuestro clasificador.

		Predicciones	
		Positivo	Negativo
Valor Real	Positivo	Verdadero Positivo	Falso Negativo
	Negativo	Falso Positivo	Verdadero Negativo

Se pueden definir las siguientes métricas de bondad para un clasificador:

- Accuracy: $\frac{VP+VN}{VP+FN+FP+VN}$.
- Preccision Score: $\frac{VP}{VP+FP}$.
- F1 score: $\frac{2VP}{2VP+FP+FN}$.
- Recall: $\frac{VP}{VP+FN}$.

El accuracy cuenta todas las veces que ha acertado el clasificador y se divide entre el total de predicciones. Preccision score el la relación entre las clasificaciones buenas de las clase positiva y todas las clasificaciones de la clase positiva.

EL recall es muy parecido pero se divide entre los valores reales positivos. Finalmente F1 es la media armónica entre preccision y recall.

Veamos los datos que obtenemos lanzando un SVM con kernel lineal sobre cada uno de los folds.

Cuadro 1: Kernel lineal, HoG

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.96	0.96	0.96	0.95	0.96
Preccison	0.96	0.97	0.95	0.96	0.97
Recall	0.93	0.96	0.94	0.95	0.95
F1	0.95	0.95	0.96	0.95	0.96

Obtenemos unos resultados bastante buenos, veamos si modificando los parámetros del clasificador podemos conseguir mejoría. Ahora a la función entrenar_clasificador() le vamos a indicar que queremos usar un kernel polinomial, mediante "POLY". Con ello haremos una transformación polinomial a un estado latente donde los datos si puedan ser separados linealmente.

```
def entrenar_clasificador(training_data, classes, kernel):

    svm = cv2.ml.SVM_create()
    svm.setType(cv2.ml.SVM_C_SVC)

    if kernel == "LINEAR":
        svm.setKernel(cv2.ml.SVM_LINEAR)
    if kernel == "POLY":
        svm.setKernel(cv2.ml.SVM_POLY)
        svm.setDegree(2)
        #svm.setDegree(3)
    if kernel == "RBF":
        svm.setKernel(cv2.ml.SVM_RBF)
        svm.setGamma(0.01)
        svm.setC(10)

    svm.train(training_data, cv2.ml.ROW_SAMPLE, classes)

    return svm
```

Los datos obtenidos son los siguientes:

Cuadro 2: Kernel polinomial grado 2, HoG

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.97	0.97	0.96	0.96	0.97
Preccison	0.98	0.98	0.96	0.96	0.98
Recall	0.95	0.96	0.96	0.95	0.95
F1	0.97	0.97	0.96	0.96	0.96

Las diferencias son insignificantes, un poco mejor el kernel polinomial aunque el tiempo de ejecución es bastante mayor. Lo mismo pasa para un polinomio de grado 3.

Cuadro 3: Kernel polinomial grado 3, HoG

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.98	0.97	0.97	0.97	0.97
Preccison	0.98	0.98	0.99	0.98	0.98
Recall	0.97	0.94	0.95	0.96	0.96
F1	0.98	0.96	0.97	0.97	0.97

Ahora probaremos con un kernel gaussiano, la función núcleo que lleva los puntos a otro espacio viene dada por $\mathbf{K}(x_i, x_j) = e^{-\gamma||x_i - x_j||^2},\ \gamma > 0$. Probaremos con $\gamma = 0{,}01$, $\gamma = 0{,}1$ y $\gamma = 1$.

Cuadro 4: Kernel Radial $\gamma = 0{,}01$, HoG

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.96	0.97	0.97	0.95	0.96
Preccison	0.95	0.96	0.97	0.92	0.96
Recall	0.97	0.98	0.96	0.97	0.97
F1	0.96	0.97	0.97	0.94	0.96

Cuadro 5: Kernel Radial $\gamma = 0{,}1$, HoG

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.92	0.92	0.91	0.93	0.92
Preccison	0.92	0.92	0.85	0.89	0.89
Recall	0.94	0.94	0.76	0.98	0.95
F1	0.93	0.93	0.91	0.93	0.92

Cuadro 6: Kernel Radial $\gamma = 1$, HoG

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.5	0.49	0.47	0.45	0.47
Preccison	1.0	0	0.47	0	0.47
Recall	0.01	0	1.0	0	1.0
F1	0.03	0	0.64	0	0.64

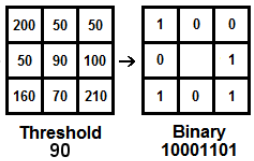
Cuanto mayor es γ más error cometemos de hecho los datos anteriores dan valores tan extremos en algunas métricas porque el clasificador a predicho todos los valores como una sola clase, ya sea positiva o negativa.

Con ello concluimos que la mejor forma para hacer una clasificación sería utilizar un kernel polinomial de grado 2 aunque el propio kernel lineal da unos resultados muy parecidos y es relativamente más rápido.

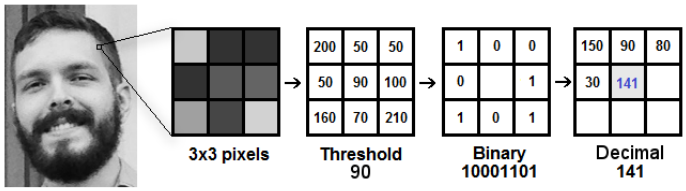
1.2. Clasificación incorporando LBP-básico.

En esta sección nos encargaremos de implementar una clase que nos calcule LBP clásico y posteriormente usaremos este descriptor para entrenar un SVM para poder clasificar si en una imagen se encuentra un peatón o no.

LBP son las siglas de Local Binary Pattern , para el computo de este descriptor debemos fijarnos en los vecinos de cada píxel de la imagen, de ahí su nombre. Para cada píxel de la imagen debemos obtener su píxel lbp, este pixel se calculo fijandonos en los 8 píxeles que hay alrededor del mismo. Compararemos el píxel central con los píxeles de su alrededor en el sentido horario, si el píxel vecino es mayor apuntamos un 1 en caso contrario 0. De tal forma como podemos observar en la imagen.



Obtenemos para cada píxel un número binario de 8 dígitos, si cambiamos a base decimal obtendremos un número comprendido entre 1 y 256. Haciendo esto para cada píxel de la imagen obtenemos una imagen de texturas lbp en escalas de grises.



Finalmente a esta imagen de texturas le calculamos el histograma, contamos cuantas veces aparecen los números comprendidos entre 1 y 256 en la imagen.

1.2.1. Implementación de LBP-básico.

El computo de LBP-clásico se hace en el script LBP.py, definimos una clase llamada LocalBinaryPattern, cuando llamamos a la clase debemos pasarle una imagen. Automáticamente pasa a escalas de grises la imagen y le añade 0 a los bordes para no tener problema a la hora de calcular los píxeles lbp de la imagen.

```python
def __init__(self, image):
    # Transformamos la imagen en escalas de grises
    self.gris = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # Añadimos en las esquinas y bordes un pixel negro
    self.gris = np.insert(self.gris, 0, np.zeros(self.gris.shape[0]), 1)
    self.gris = np.insert(self.gris, self.gris.shape[1], np.zeros(self.gris.shape[0]), 1)
    self.gris = np.insert(self.gris, self.gris.shape[0], np.zeros(self.gris.shape[1]), 0)
    self.gris = np.insert(self.gris, 0, np.zeros(self.gris.shape[1]), 0)
```

Ahora definimos un método que nos devuelva el descriptor de la imagen que le hayamos pasado a la clase. En compute_lbp_clasic() calculamos matricialmente la imagen de texturas lbp. Una primera idea para calcular los píxeles es ir uno por uno y compararlo con sus 8 vecinos, el problema es el tiempo de computo. Python es un lenguaje de programación no compilado por lo que la implementación de bucles for aumenta mucho el tiempo de ejecución.

Vamos a calcular 8 matrices usando la matriz con bordes negros y en escala de grises, tomaremos la submatriz sin los bordes, es decir nuestra imagen original. Esta matriz la vamos a comparar con la submatriz desplazada una columna a la izquierda y una fila hacia arriba, haremos el proceso 7 veces más con submatrices desplazadas al redecor de la imagen en el sentido original. Por último sumaremos todas estas matrices con sus respecivas potencias de dos para obtener la imagen de texturas.

```python
def compute_lbp_clasic(self):

    n_row = np.shape(self.gris)[0] - 1
    n_col = np.shape(self.gris)[1] - 1

    M0 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1-1:n_row-1, 1-1:n_col-1], dtype="float32")*2**7
    M1 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1-1:n_row-1, 1:n_col], dtype="float32")*2**6
    M2 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1-1:n_row-1, 1+1:n_col+1], dtype="float32")*2**5
    M3 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1:n_row, 1+1:n_col+1], dtype="float32")*2**4
    M4 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1+1:n_row+1, 1+1:n_col+1], dtype="float32")*2**3
    M5 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1+1:n_row+1, 1:n_col], dtype="float32")*2**2
    M6 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1+1:n_row+1, 1-1:n_col-1], dtype="float32")*2
    M7 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1:n_row, 1-1:n_col-1], dtype="float32")*1
    # Matriz con valor lbp de cada pixel
    M = M0 + M1 + M2 + M3 + M4 + M5 + M6 + M7
```

En lugar de hacer directamente el histograma sobre toda la imagen, tomaremos submatrices de 16 por 16 y desplazadas 8 píxeles, a estas le calcularemos el histograma y los concatenaremos para dar resultado a un descriptor final de 105*256 variables (105 ventanas y 256 características de cada histograma).

```python
descriptor = []
hist = [M[row:row+16,col:col+16] for row in range(0,n_row-16,8) for col in range(0, n_col-16,8)]

for ventanita in hist:
        histogramilla = cv2.calcHist([ventanita], [0], None, [256], [0, 256])
        histogramilla = cv2.normalize(histogramilla, histogramilla)
        descriptor.append(histogramilla)
descriptor = np.array(descriptor,dtype="float32").reshape(26880,1)

return descriptor
```

Cada uno de los histogramas realizados en las ventanitas de 16*16 es normalizado, así el descriptor es más robusto ante cambios de exposición en las imágenes.

1.2.2. Uso de LBP-clásico para clasificación.

Usaremos la clase LocalBinaryPattenr para el computo del descriptor que utilizaremos para entrenar y testear el clasificador. Modificaremos el script cargar_imagenes_datos.py para calcular el descriptor lbp clásico.

```
for filename in os.listdir(PATH_POSITIVE):
    if filename.endswith(IMAGE_EXTENSION):
        filename = PATH_POSITIVE+filename
        img = cv2.imread(filename)
        #hog = cv2.HOGDescriptor()
        #descriptor = hog.compute(img)
        lbp = LBP.LocalBinaryPattern(img)
        descriptor = lbp.compute_lbp_clasic()
        data.append(descriptor)
        clases.append(1)
        counter_positive_samples += 1
        print("Leidas " + str(counter_positive_samples) + " imágenes de -> peatones")
```

Al igual que hicimos en la primera sección, usaremos distintos kernels y parámetros para entrenar el clasificador y veremos que kernels y parámetros son los óptimos para clasificar con lbp clásico.

Cuadro 7: Kernel lineal, LBP-clásico.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9958	0.9958	0.9916	1.0	0.9958
Preccison	0.9921	1.0	0.9830	1.0	0.9915
Recall	1.0	0.9915	1.0	1.0	1.0
F1	0.9960	0.9957	0.9914	1.0	0.9957

Cuadro 8: Kernel polinomial grado 2, LBP-clásico.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9916	0.9916	0.9916	1.0	0.9916
Preccison	0.9838	0.9846	0.9823	1.0	0.9838
Recall	1.0	1.0	1.0	1.0	1.0
F1	0.9918	0.9922	0.9910	1.0	0.9918

Cuadro 9: Kernel polinomial grado 3, LBP-clásico.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9875	0.9958	1.0	0.9875	1.0
Preccison	0.9741	0.9921	1.0	0.9834	1.0
Recall	1.0	1.0	1.0	1.0	1.0
F1	0.9868	0.9922	1.0	0.9875	1.0

Cuadro 10: Kernel radial $\gamma = 0{,}01$, LBP-clásico.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	1.0	0.9958	0.9958	0.9958	0.9708
Preccison	1.0	0.9916	0.9913	0.992	0.9743
Recall	1.0	1.0	1.0	1.0	0.9661
F1	0.9958	0.9956	0.99459	0.9956	0.9661

Cuadro 11: Kernel radial $\gamma = 0{,}1$, LBP-clásico.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9	0.94	0.95	0.93	0.83
Preccison	0.82	0.91	0.91	0.88	0.77
Recall	1.0	0.98	1.0	0.99	1.0
F1	0.9055	0.9509	0.9531	0.9370	0.8712

Cuadro 12: Kernel radial $\gamma = 1$, LBP-clásico.

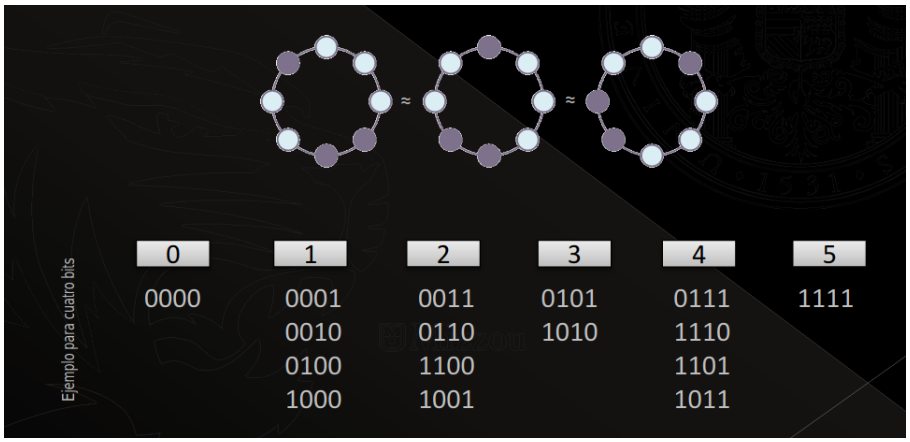
	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.48	0.49	0.46	0.46	0.48
Preccison	0	0	0.46	0	0.48
Recall	0	0	1	0.99	0.65
F1	0	0.9509	0.63	0.	1.0

Mejoramos con respecto el descriptor HoG, el comportamiento de los clasificadores con los distintos parámetros es muy parecido a lo que ocurría con HoG. El kernel Gausiano empeora aumentando γ, el lineal y polinomial serían los más óptimos para este problema.

2. Mejoras Optativas.

2.1. LBP-uniforme.

Otra versión para lbp, es la denominada lbp-uniformes, que puede ser usado para reducir la longitud del vector de características y para implementar un descriptor sencillo e invariante frente a rotaciones. Esta versión fue inspirada porque algunos patrones binarios son más frecuentes en las imágenes de textura que otros. Un código LBP se llama homogéeneo si el patrón binario contiene un máximo de dos transiciones a nivel de bits, de 0 a 1 o viceversa, cuando el patrón de bits es atravesado de manera circular. Por ejemplo, los patrones de 00000000 (0 transiciones), 01110000 (2 transiciones) y 11001111 (2 transiciones) son uniformes, mientras que los patrones de 11001001 (4 transiciones) y 01010010 (6 transiciones) no lo son. En el cálculo de las etiquetas LBP, cuando los patrones uniformes son utilizados, se utiliza una etiqueta para cada uno de los patrones uniforme y todos los patrones no uniformes están etiquetados con una sola etiqueta. Por ejemplo, cuando se utilizan los 8 vecinos más cercanos, hay un total de 256 patrones, 58 de los cuales son uniformes y el resto son no uniformes, por lo que resulta un total de 59 etiquetas diferentes.



2.1.1. Implementación de LBP-uniforme.

Para la implementación de esta variante de lbp definimos otro método en la clase LocalBinaryPattern. Al igual que hicimos para lbp clásico calculamos la matriz de texturas lbp y construimos ventanitas de 16*16 en saltos de 8 píxeles creando un total de 105 submatrices. En una lista guardamos todos los códigos LBP uniformes posibles, teniendo así una lista de 58 elementos. Esta conjunto de números se puede conseguir fácilmente en Wikipedia; por ejemplo. Entonces los píxeles de la imagen de texturas que tengan un valor distinto a los de la lista representan píxeles no homogéneos. Formaremos dos histogramas, uno con los 256 patrones de lbp clásico y otro con 59 patrones. En el segundo

histograma vamos a usar los valores del primer histograma asociado a los códigos lbp-uniforme y en el ultimo patrón vamos a poner la suma de todos los otros códigos que no son uniformes.

```
def compute_lbp_uniform(self):

    n_row = np.shape(self.gris)[0] - 1
    n_col = np.shape(self.gris)[1] - 1

    M0 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1-1:n_row-1, 1-1:n_col-1], dtype="float32")*2**7
    M1 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1-1:n_row-1, 1:n_col], dtype="float32")*2**6
    M2 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1-1:n_row-1, 1+1:n_col+1], dtype="float32")*2**5
    M3 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1:n_row, 1+1:n_col+1], dtype="float32")*2**4
    M4 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1+1:n_row+1, 1+1:n_col+1], dtype="float32")*2**3
    M5 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1+1:n_row+1, 1:n_col], dtype="float32")*2**2
    M6 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1+1:n_row+1, 1-1:n_col-1], dtype="float32")*2
    M7 = np.array(self.gris[1:n_row, 1:n_col] >= self.gris[1:n_row, 1-1:n_col-1], dtype="float32")*1
    # Matriz con valor lbp de cada pixel
    M = M0 + M1 + M2 + M3 + M4 + M5 + M6 + M7

    descriptor = []
    hist = [M[row:row+16,col:col+16] for row in range(0,n_row-16,8) for col in range(0, n_col-16,8)]
    #print(hist[100])
    patterns = [0, 1, 2, 3, 4, 6, 7, 8, 12, 14, 15, 16, 24, 28, 30, 31, 32, 48, 56, 60, 62, 63, 64, 96, 112,
    120, 124, 126, 127, 128, 129, 131, 135, 143, 159, 191, 192, 193, 195, 199, 207, 223, 224, 225, 227, 231,
    239, 240, 241, 243, 247, 248, 249, 251, 252, 253, 254, 255]

    for ventanita in hist:
            histogramilla = cv2.calcHist([ventanita], [0], None, [256], [0, 256])
            histogramilla_u = np.zeros((59,))
            histogramilla_u[:58]= histogramilla[patterns].reshape(58,)
            histogramilla_u[58] = np.sum(histogramilla[[i for i in range(256) if i not in patterns]])
            histogramilla_u = cv2.normalize(histogramilla_u, histogramilla_u)
            descriptor.append(histogramilla_u)

    descriptor = np.array(descriptor,dtype="float32").reshape(6195,1)
```

2.1.2. LBP-uniforme para clasificación de peatones.

Volveremos a repetir el proceso de clasificación ahora con el descriptor dado por lbp-uniforme.

Cuadro 13: Kernel lineal, LBP-unifomre.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9916	0.9916	0.9958	0.9875	0.9875
Preccison	0.9837	0.9823	0.9923	0.9831	0.9915
Recall	1.0	0.9915	1.0	0.9915	0.98
F1	0.9918	0.9910	0.9961	0.9873	0.9874

Cuadro 14: Kernel polinomial grado 2, LBP-unifomre.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9958	0.9958	1.0	1.0	0.9916
Preccison	0.9913	0.9918	1.0	1.0	0.9833
Recall	1.0	1.0	1.0	1.0	1.0
F1	0.9956	0.9959	1.0	1.0	0.9915

Cuadro 15: Kernel polinomial grado 3, LBP-unifomre.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	1.0	0.9958	0.9875	0.9916	0.9791
Preccison	1.0	0.9911	0.9763	1.0	0.9583
Recall	1.0	1.0	1.0	0.9838	1.0
F1	1.0	0.9804	0.9880	0.9918	0.9787

Cuadro 16: Kernel radial $\gamma = 0{,}01$, LBP-unifomre.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	1.0	0.9958	0.9875	0.9916	0.9791
Preccison	1.0	0.9911	0.9763	1.0	0.9583
Recall	1.0	1.0	1.0	0.9838	1.0
F1	1.0	0.9804	0.9880	0.9918	0.9787

Cuadro 17: Kernel radial $\gamma = 0{,}1$, LBP-unifomre.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9625	0.9666	0.9458	0.9666	0.9666
Preccison	0.9398	0.9541	0.9069	0.9316	0.9444
Recall	0.9920	0.9842	0.9915	1.0	0.9916
F1	0.9652	0.9689	0.9473	0.9646	0.9674

Cuadro 18: Kernel radial $\gamma = 1$, LBP-unifomre.

	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.5458	0.4708	0.4958	0.4666	0.4916
Preccison	0.5240	0.4708	0	0	0.4916
Recall	1.0	1.0	0	0	1.0
F1	0.6876	0.6402	0	0	0.6592

Los datos obtenidos son muy parecidos a los que teníamos en lbp-clásico. Los parámetros que optimizan el modelo son kenerel lineal o polinomial. Como resumen podríamos decir que las ventajas del lbp-uniforme frente al clásico es la invarianza frente rotaciones y que el descriptor tiene menos características, así conseguimos que el clasificador sea más rápido para entrenar y además el descriptor es más robusto.

2.2. Combinación de características.

Combinaremos las características de HoG y LBP para entrenar el clasificador. Compararemos con los casos anteriores y comprobaremos si mejoramos los resultados.

Para construir el nuevo descriptor combinando las características lo único que vamos a hacer es concatenar HoG y lbp, usaremos lbp-clásico.

```
img = cv2.imread(filename)
hog = cv2.HOGDescriptor()
descriptor_1 = hog.compute(img)
lbp = LBP.LocalBinaryPattern(img)
descriptor_2 = lbp.compute_lbp_clasic()
#descriptor = lbp.compute_lbp_uniform()
descriptor = np.concatenate((descriptor_1, descriptor_2))
```

Este nuevo descriptor será el que usemos para hacer la clasificación. Veamos los resultados que obtenemos al hacer validación cruzada con un tuneo de los parámetros. Por último resumiremos que descriptor y que parámetros son los mejores para nuestro modelo de clasificación de peatones.

Hemos notado leves mejorías frente a los otros descriptores, en esta ocasión un kernel lineal es lo mejor para la clasificación. En la siguiente tabla podemos observar los resultados de cada uno de los modelos con distintos parámetros. Como conclusión frente a todos los descriptores y parámetros que hemos usado, podemos tomar HoG + LBP-clásico y svm con kernel lineal como el mejor modelo para clasificar peatones.

Kernel lineal, LBP clasico + HoG					
	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	1.0	1.0	0.9958	1.0	0.9916
Preccison	1.0	1.0	0.992	1.0	0.9837
Recall	1.0	1.0	1.0	1.0	1.0
F1	1.0	1.0	0.995	1.0	0.9918
Kernel polinomial grado 2, LBP clasico + HoG					
	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9958	0.9916	1.0	0.9958	0.9958
Preccison	0.9918	0.9833	1.0	0.9918	0.9919
Recall	1.0	1.0	1.0	1.0	1.0
F1	0.9958	0.9915	1.0	0.9958	0.9959
Kernel polinomial grado 3, LBP clasico + HoG					
	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9916	1.0	0.9958	0.9916	1.0
Preccison	0.9843	1.0	0.9924	0.9829	1.0
Recall	1.0	1.0	1.0	1.0	1.0
F1	0.9921	1.0	0.9961	0.9913	1.0
Kernel radia l$\gamma = 0{,}01$, LBP clasico + HoG					
	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.9916	0.9916	1.0	0.9916	0.9916
Preccison	0.9923	0.9839	1.0	0.9821	0.9830
Recall	0.9923	1.0	1.0	1.0	1.0
F1	0.9923	0.9914	1.0	0.9909	0.9914
Kernel radia l$\gamma = 0{,}1$, LBP clasico + HoG					
	Fold1	Fold2	Fold3	Fold4	Fold5
Accuracy	0.5500	0.4375	0.5541	0.6350	0.4375
Preccison	1.0	0.4375	0.5244	0.5776	1.0
Recall	0.1074	1.0	1.0	1.0	0.014
F1	0.1940	0.6086	0.6880	0.7323	0.0287

2.3. Localización de peatones a diferentes escalas.

En los ejemplos anteriores hemos trabajado con imágenes 128x64 donde había una sola persona o fondo, si bien no se ha aplicado al caso de imágenes de mayor tamaño con personas a diferentes escalas. En este apartado usaremos el mejor modelo de los obtenidos hasta ahora para localizar un peatón en una imagen dada de cualquier tamaño.

Para la detección de personas en una imagen vamos a usar la función personas_hog_lbp() definida en deteccion_de_personas.py, cuando le pasemos una imagen a la función esta nos devolverá una lista donde está almacenada la posición de cada peatón detectado. Veamos como funciona más detalladamente.

```
def personas_HOG_LBP(img):
    # Variables auxiliares
    cantidad = 0
    peatones = [] # Peaton detectado
    alto_maximo = img.shape[0]
    ancho_maximo = img.shape[1]
    for alto in np.arange(0, alto_maximo - 128, 32):
        for ancho in np.arange(0, ancho_maximo - 64, 32):
```

Vamos a recorrer la imagen con pequeñas ventanas de 128*64, cada 32 píxeles, que es justo la medida de las imágenes con las que hemos entrenado el clasificador. A cada una de estas imágenes le calcularemos su descriptor HoG+LBP y predecimos si es peatón o no mediante el svm previamente entrenado.

```
# Calculo el hog y lbp
hog = cv2.HOGDescriptor()
h_1 = hog.compute(ventanita)
lbp = LBP.LocalBinaryPattern(ventanita)
h_2 = lbp.compute_lbp_clasic()
h = np.concatenate((h_1,h_2))
# Lo paso a 1 dimension
h2 = h.reshape((1, -1))
# Pruebo si hay un peaton
pred = ejercicio1.svm_modelo.predict(h2)[1]
```

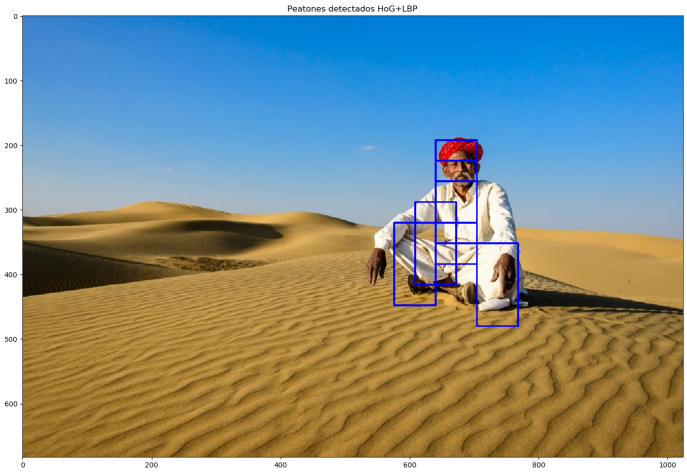
Una vez hayamos calculado la predicción para esa ventana, en caso de que hayamos predicho peatón guardaremos en una lista los valores de su posición.

```
# Si predecimos peaton
if pred == 1:
    # Guardo la posicion de la deteccion para graficar
    detectado = [] # Genero lista vacia
    detectado.append(ancho)
    detectado.append(alto)
    detectado.append(ancho+64)
    detectado.append(alto+128)
    peatones.append(detectado) # Lo agrego a la lista general
    cantidad += 1              # Cuento un peaton más
    print(cantidad)
```

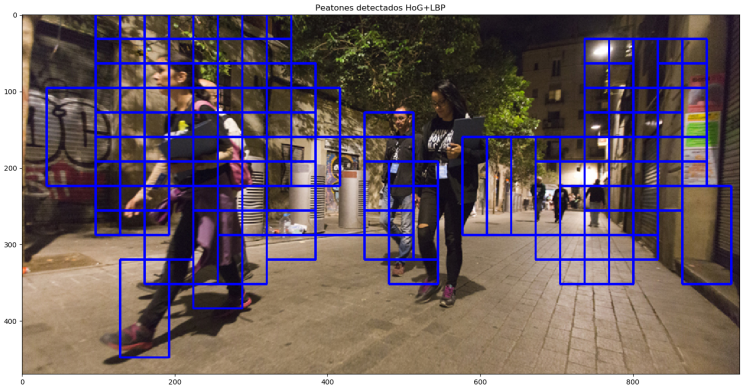
Por último probamos la función con distintas imágenes y las mostramos por pantalla.

```
imagen = cv2.imread("4.jpg")
peatones_encontrados = personas_HOG_LBP(imagen)
copia_hog = imagen.copy()

for peatonf in peatones_encontrados:
    v1 = int(peatonf[0])
    v2 = int(peatonf[1])
    v3 = int(peatonf[2])
    v4 = int(peatonf[3])
    cv2.rectangle(copia_hog, (v1, v2), (v3, v4), (255,0,0), 2)

img_peatones = copia_hog[:,:,::-1]
fig = plt.figure(figsize = (15, 15))
ax = fig.add_subplot(111)
ax.imshow(img_peatones, interpolation='none')
ax.set_title('Peatones detectados HoG+LBP')
plt.show()
```

Obteniendo así los siguientes resultados:



El mayor problema que tenemos son los falsos positivos como vemos a continuación:



Por lo general detecta bien a las personas pero detecta otras muchas como peatones que no lo son.