Tarea 3:
Clasificación de edad usando características tipo HOG

Fecha de entrega: Miércoles 13 de octubre, 23:59 hrs.

Estudiante: Francisco Molina L.
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla Z.
Semestre: Primavera 2021

Índice

1. Introducción

En el presente informe se desarrollan las actividades correspondientes a la Tarea 3 del curso EL7008-1 del semestre de primavera de 2021. Este informe comprende la implementación computacional de la extracción de características HOG, y del entrenamiento y análisis de clasificadores que clasifican personas según su edad.

El principal objetivo de este informe es lograr implementar mediante programación el cálculo de las características HOG, lo que implica re-dimensionar, calcular gradientes (su magnitud y su orientación) y calcular histogramas de la orientación de los gradientes. Los clasificadores no se programan a bajo nivel, sino que simplemente se importan desde la librería `sklearn` y se analiza su desempeño cuando se entrenan con las características HOG programadas.

La **Sección 2** contiene un resumen del contexto necesario para entender las implementaciones realizadas en la tarea, esto incluye las características HOG y una explicación detrás del funcionamiento de las *Support Vector Machines* y de los *Random Forest Classifiers* según [2]. En las **Secciones 3 y 4** se detallan las funciones implementadas en Python, adjuntas en el notebook Tarea3.ipynb, necesarias para calcular las características HOG, realizar el entrenamiento de los clasificadores y realizar las predicciones sobre los conjuntos de datos. En la **Sección 5** se presentan los resultados obtenidos de la implementación en Python, en la **Sección 6** se realiza un análisis de dichos resultados, destacando los casos interesantes, y en la **Sección 7** se presentan las conclusiones del informe.

2. Marco Teórico

2.1. Características HOG

El primer paso para realizar reconocimiento de clases de objetos es extraer las características de los objetos. Las características escogidas para esta tarea corresponden al Histograma de Gradientes Orientados (HOG), el cual se obtiene mediante los pasos que se describen a continuación.

En primer lugar, se transforma la imagen a escala de grises y se transforma su tamaño a (64,128), como se muestra en la **Figura 1**:

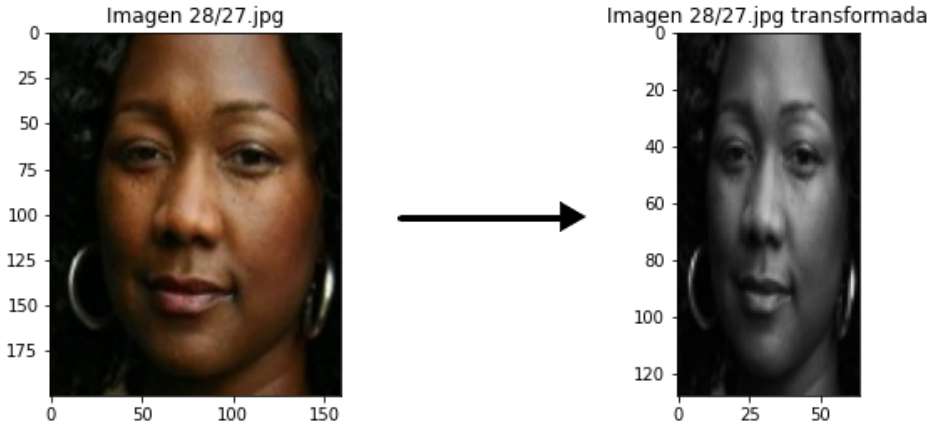

Figura 1: Ejemplo de transformación de imagen

Se calculan los gradientes de la imagen transformada, en la dirección horizontal (G_x) y la vertical (G_y), como se muestra en la **Figura 2**:

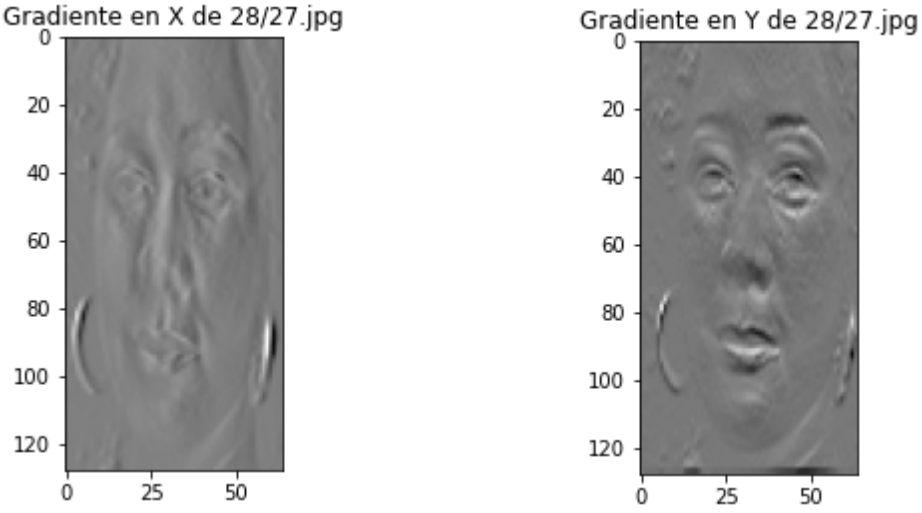

Figura 2: Ejemplo de cálculo de gradientes

Y en base a estos gradientes se calculan las matrices de magnitud y orientación de los gradientes para cada posición [y,x] de la imagen transformada, como se muestra en la **Figura 3**:

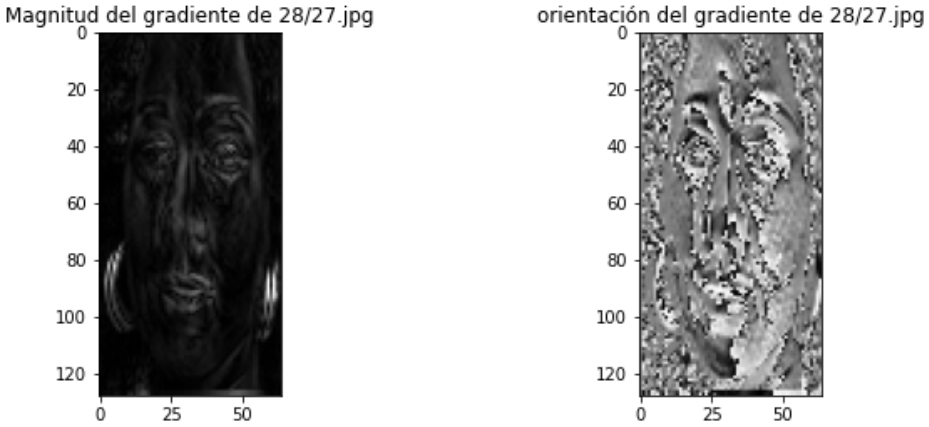

Figura 3: Ejemplo de matrices de magnitud y orientación

La magnitud y la orientación del gradiente para cada posición [y,x] se obtienen mediante:

$$\begin{aligned} mag(y,x) &= \sqrt{G_x(y,x)^2 + G_y(y,x)^2} \\ ang(y,x) &= arctan(\frac{G_y(y,x)}{G_x(y,x)}) \end{aligned} \tag{1}$$

Ya habiendo calculado las matrices de magnitud y orientación del gradiente de la imagen, se divide la imagen transformada en $8x16$ celdas de tamaño (8,8) cada una, como se muestra en la **Figura 4**:

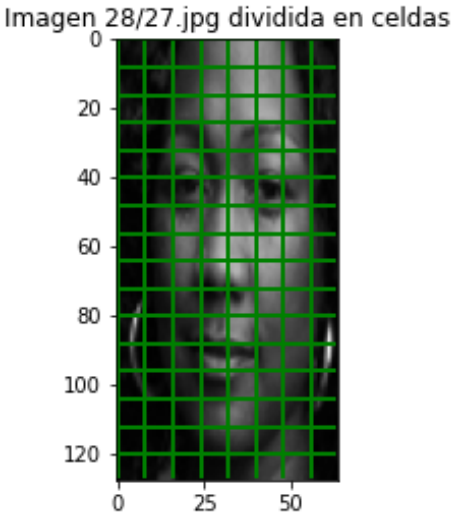

Figura 4: Ejemplo de división de imagen en celdas

Para cada una de las celdas se calcula el histograma de su gradiente, donde cada posición vota en el bin correspondiente a la orientación de su gradiente y suma tantos votos como la magnitud de su gradiente. De manera arbitraria se escoge que el histograma tendrá 9 bines, por lo que los rangos de orientaciones correspondientes a cada bin son como se muestra en la **Figura 5:**

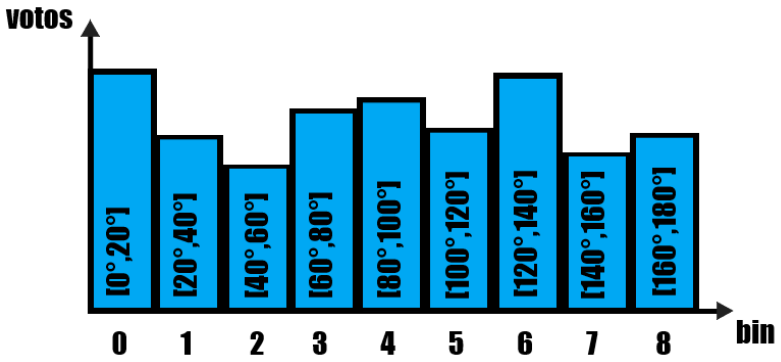

Figura 5: Ejemplo de histograma para una celda

Dada esta distribución, un pixel con orientación de gradiente θ vota en el bin dado por:

$$\lfloor \frac{\theta}{20^\circ} \rfloor \tag{2}$$

Este pixel suma tantos votos como la magnitud de su gradiente μ, pero esta se pondera en función de la distancia del pixel al centro de la celda, de modo que los pixeles que están más cerca del centro tienen un voto más importante que el resto. Entonces el voto registrado al bin $\lfloor \frac{\theta}{20^\circ} \rfloor$ corresponde a:

$$bin[\lfloor \frac{\theta}{20^\circ} \rfloor] \stackrel{+}{=} \mu \cdot dist((y,x),(y_c,x_c)) \tag{3}$$

Una vez se hace esto para todas las celdas, el último paso consiste en realizar una normalización por bloques para que los HOG sean invariantes a los cambios de iluminación y contraste. Este procedimiento consiste en tomar bloques de (2,2) celdas, concatenar sus histogramas y normalizarlos. Las 4 celdas se escogen en forma de cuadrado y con traslape, como se muestra en la **Figura 6**:

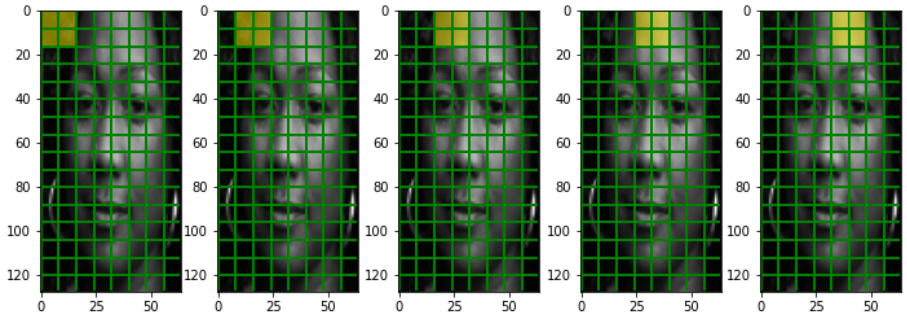

Figura 6: Ejemplo de block normalization

Una vez obtenidos todos los histogramas normalizados de todos los bloques, estos se concatenan para formar el vector de características HOG.

Dado que cada celda tiene un histograma de 9 bines, y los bloques se forman tomando 4 celdas, cada bloque forma un vector de características de 36 posiciones. Dado que los bloques se forman con las celdas vecinas (a la derecha y abajo), se deben omitir las celdas de las últimas filas y columnas, pues no tienen más vecinos. Como hay (7,15) celdas en las que se pueden formar bloques, y se concatenan todos los histogramas de 36, entonces el vector de características HOG es de tamaño:

$$len(\text{HOG}) = 9 \cdot 4 \cdot 7 \cdot 15 = 3780 \tag{4}$$

2.2. Clasificadores

Una vez obtenidas las características HOG, lo siguiente es entrenar los clasificadores usando como entrada las características HOG. En esta sección se describen brevemente los clasificadores que se utilizan en esta tarea, que corresponden a Support Vector Machines y Random Forest.

2.2.1. Support Vector Machines

Los clasificadores SVM están basados en la clase de hiperplanos:

$$(w \cdot x) + b = 0, \quad w \in \mathbb{R}^d, b \in \mathbb{R}^d \tag{5}$$

Correspondientes a las funcinoes de decisión:

$$f(x) = sgn((w \cdot x) + b) \tag{6}$$

Entre todos los hiperplanos que separan los datos, existe un único **hiperplano óptimo** que otorga el **máximo margen de separación entre las clases**:

$$\max_{w,b} min\{||x - x_i|| : x \in \mathbb{R}^d, (w \cdot x) + b = 0, i = 1, ..., n\} \tag{7}$$

Lo que gráficamente corresponde a lo mostrado en la **Figura 7**:

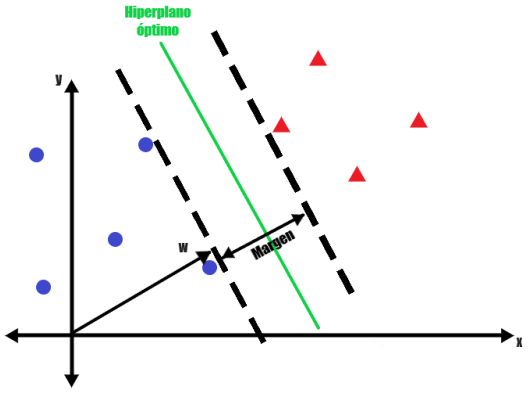

Figura 7: Visualización de hiper-plano óptimo en 2D

Se definen fronteras al hiper-plano óptimo como rectas paralelas al hiper-plano óptimo que pasan por los puntos más cercanos a él, y estas deben satisfacer que:

$$\begin{aligned}(w \cdot x_1) + b &= +1 \\ (w \cdot x_2) + b &= -1\end{aligned} \tag{8}$$

Tomando la diferencia de estas ecuaciones se obtiene:

$$\begin{aligned}(w \cdot (x_1 - x_2)) &= 2 \;\; / \;\; \cdot \frac{1}{||w||} \\ \frac{w}{||w||} \cdot (x_1 - x_2) &= \frac{2}{||w||}\end{aligned} \tag{9}$$

Aquí $\frac{w}{||w||}$ corresponde al vector unitario con dirección perpendicular al hiper-plano óptimo, por lo que $\frac{w}{||w||} \cdot (x_1 - x_2)$ corresponde a la proyección de la distancia entre x_1 y x_2 sobre el eje perpendicular al plano, es decir, el **margen** se puede computar mediante el inverso de la norma del vector de pesos.

Si se parte por el supuesto de que el conjunto de entrenamiento efectivamente puede ser separado por el hiper-plano:

$$D(x) = (w \cdot x + b), \quad w \in \mathbb{R}^d, b \in \mathbb{R} \tag{10}$$

El cual debe satisfacer la siguiente restricción:

$$y_i[(w \cdot x_i) + b] \geq 1, \quad i = 1, ..., n \tag{11}$$

Se llama margen (τ) a la distancia mínima del hiper-plano separador al punto de dato más cercano, un hiper-plano separador se dice óptimo si el margen es máximo.

Se puede ver que para un punto x que se encuentra dentro del hiper-plano separador se tiene:

$$D(x) = (w \cdot x) + b = 0 \tag{12}$$

Mientras que para un punto x' fuera del hiper-plano se tiene que:

$$D(x') = (w \cdot x') + b \tag{13}$$

Restando ambas expresiones:

$$\begin{aligned}D(x') = w \cdot (x' - x) \;\; / \;\; \cdot \frac{1}{||w||} \\ \frac{w}{||w||} \cdot (x' - x) = \frac{D(x')}{||w||}\end{aligned} \tag{14}$$

Lo que corresponde a la distancia del punto x' al hiper-plano. Suponiendo que existe un margen τ, para todos los patrones se cumple la siguiente desigualdad:

$$\frac{y_k D(x_k)}{||w||} \geq \tau, \quad k = |, ..., n \tag{15}$$

Entonces, el problema de hallar el hiper-plano óptimo corresponde a encontrar el w que maximiza el margen τ. Como hay un número infinito de soluciones que difieren sólo en el escalamiento de w, se limita a una única solución fijando la escala, de maner arbitraria, a:

$$\tau||w|| = 1 \tag{16}$$

Maximizar el margen τ es equivalente a minimizar la norma de w, es decir, un hiper-plano óptimo es aquel que minimiza:

$$\eta(w) = ||w||^2 \tag{17}$$

con respecto a w y b, sujeto a la restricción $y_i[(w \cdot x_i) + b] \geq 1$.

Finalmente, para encontrar el hiper-plano óptimo basta con resolver el siguiente problema de optimización con restricciones:

$$\begin{aligned} min \;\; & \tau(w) = \frac{||w||^2}{2} \\ s.a \;\; & y_i[(w \cdot x_i) + b] \geq 1, \;\; i = 1, ..., n, \; y_i \in \{-1, +1\} \end{aligned}$$

En la práctica se resuelve el problema de optimización dual puesto que esta versión es muy lenta para problemas de alta dimensionalidad. También se relaja el margen con el fin de encontrar soluciones óptimas en las que puede haber elementos de otras clases en lados equivocados del hiper-plano óptimo.

Por lo que el problema que realmente se resuelve es de la forma:

$$\begin{aligned} max \;\; & w(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j (x_i \cdot x_j) \\ s.a \;\; & \sum_{i=1}^{n} y_i \alpha_i = 0, \;\; 0 \leq \alpha_i \leq \frac{C}{n}, \;\; i = 1, ..., n \end{aligned}$$

Independiente de la explicación algebraica recién presentada, las SVM simplemente buscan un hiper-plano óptimo de separación entre las clases de datos, donde hay tres hiper-parámetros importantes que hay que seleccionar: C que corresponde al coeficiente de relajación del margen, *kernel* que corresponde al kernel que se va a utilizar en el truco del kernel para resolver el problema de las SVM no lineales, y *gamma* que corresponde al coeficiente que se utiliza con los kernels: lineal, poly y rbf.

2.2.2. Random Forest

Los bosques aleatorios corresponden a una estructura que combina las salidas de múltiples árboles de decisión (*Decision Trees*) formados con datos escogidos al azar.

Antes de explicar los RF, es necesario explicar los árboles de decisión. Un árbol es una estructura que tiene tres tipos de nodos:

- Un nodo raíz que no tiene arcos entrantes, sólo tiene arcos salientes.
- Nodos internos, cada uno de los cuales tiene un arco entrante y dos o más arcos salientes.
- Nodo hoja o terminales, cada uno de los cuales tiene un arco entrante.

La estructura de un árbol se puede visualizar como se presenta en la **Figura 8**:

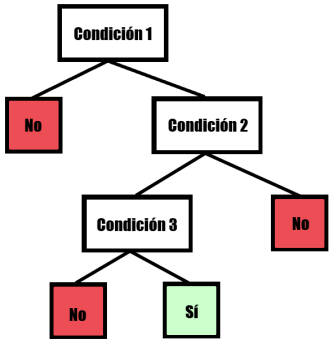

Figura 8: Ejemplo de árbol de decisión

A cada nodo de hoja se le asigna una etiqueta de clase. Los nodos no terminales, que incluyen la raíz y otros nodos internos, contienen pruebas sobre los atributos que separan los datos que presenten resultados diferentes para dichos atributos. En esencia, el árbol de decisión fragmenta el conjunto de datos de manera recursiva hasta asignar los ejemplos a las clases.

Visualmente, un árbol de decisión hace cortes perpendiculares a los ejes, como se puede ver en la **Figura 9**:

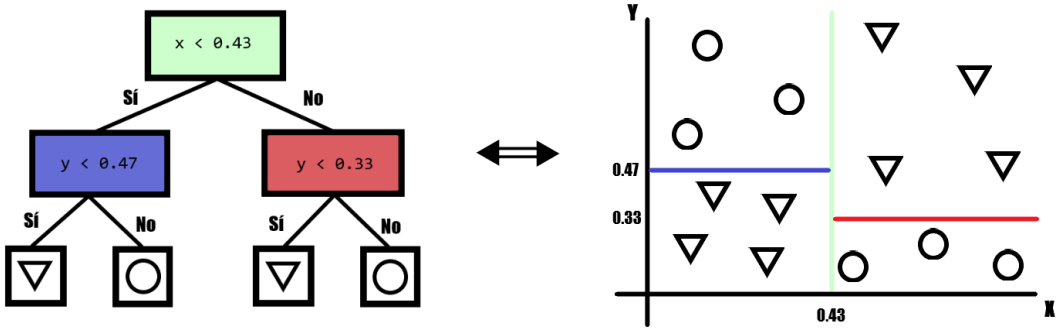

Figura 9: Espacio de parámetros dividido por árbol de decisión

Normalmente un árbol de decisión se entrena con todos los datos del conjunto de entrenamiento y los atributos que se escogen para dividir el conjunto son aquellos que maximizan una función de información. Sin embargo, los RF crean múltiples árboles de decisión, con sub-conjuntos aleatorios de los datos y con parámetros escogidos al azar, y toman las decisiones promediando las predicciones individuales de cada uno de sus árboles de decisión.

En la práctica, RF divide de manera recursiva el espacio de los datos mediante muchos árboles de decisión y promedia sus salidas para tomar decisiones. Los hiper-parámetros relevantes que hay que seleccionar son: $n\_estimators$ que corresponde a la cantidad de árboles de decisión que se forman, *criterion* que corresponde a la función que evalúa la calidad de una división realizada por una prueba, y max_depth que corresponde a la profundidad máxima que pueden tener los árboles de decisión.

3. Extracción de características HOG

3.1. Preparar conjuntos de entrenamiento, prueba y validación

Se deben separar las 600 imágenes entregadas por el equipo docente, las cuales incluyen 200 imágenes de personas entre 1 a 4 años, 200 imágenes de personas entre 5 a 27 años y 200 imágenes de personas mayores a 28 años, en los conjuntos de entrenamiento, prueba y validación.

La proporción indicada para la separación es del 60 % de las imágenes en el conjunto de entrenamiento, un 20 % en el conjunto de validación y el otro 20 % en el conjunto de prueba.

Para esto se decide almacenar las rutas de las imágenes (de la forma `carpeta/número.jpg`) en una lista `X`, y se almacenan las clases de las imágenes en una lista `y`. Donde las clases de las imágenes se corresponden a sus carpetas: 0 para las personas entre 1 a 4 años, 1 para las personas entre 5 a 27 años, y 2 para las personas mayores a 28.

Una vez obtenidas las listas `X` e `y`, estas se dividen en los conjuntos indicados mediante la función `train_test_split` de `sklearn`. Para lograr las proporciones deseadas basta con aplicar dos veces la función, la primera dejando el 20 % de los datos para el conjunto de prueba, y la segunda repartiendo el resto, como se muestra en el **Listing 1**:

```
# dividir train 80 % y test 20 %
X_train, X_test, y_train, y_test  = train_test_split(X, y, test_size=0.2,
    random_state=1)

# dividir train 60 % (0.75*0.8) y val 20 % (0.25*0.8)
X_train, X_val, y_train, y_val  = train_test_split(X_train, y_train, test_size
    =0.25, random_state=1)
```

Listing 1: repartir datos

3.2. Transformar a escala de grises y redimensionar

Como se explicó en la **Sección 2.1**, el primer paso para extraer las características HOG corresponde a transformar la imagen de entrada a escala de grises y cambiar su tamaño a (64,128).

La función encargada de esta tarea es `grayAndResize(img)` que recibe como entrada la imagen que se quiere transformar `img`. Si la imagen tiene canales de color entonces su vector de `shape` será de tamaño mayor a 2, en cuyo caso la imagen se transforma a escala de grises. Luego se redimensiona la imagen utilizando la función `resize` de `cv2`, como se muestra en el **Listing 2**:

```
def grayAndResize(img):
  out = np.copy(img)
  if len(img.shape) > 2:
    out = cv2.cvtColor(out,cv2.COLOR_RGB2GRAY)
  out = cv2.resize(out, (64,128)).astype(np.float32)
  return out
```

Listing 2: Transformar a escala de grises y redimensionar

3.3. Gradientes en x y en y

Se reutilizan las funciones creadas para calcular los gradientes en x y en y hechas en la Tarea 2, sin embargo, se modifica el kernel que utilizan. Según [4], el operador Sobel (utilizado en la Tarea 2) empeora el rendimiento de los detectores de rostros, por lo que es preferible utilizar kernels de la forma:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \tag{1}$$

Las funciones que calculan los gradientes son `gradx(img)` y `grady(img)`, donde ambas reciben como entrada la imagen a la cual se le quiere calcular el gradiente `img`. Estas funciones simplemente realizan una convolución entre la imagen de entrada con su respectivo kernel (G_x) o G_y), lo que en el código se traduce a:

```
def gradx(input):
  Gx = np.array([[-1, 0 ,1]], dtype = np.float32)
  return convolution_cython(input, Gx)

def grady(input):
  Gy = np.array([[-1],
                 [0],
                 [1]], dtype = np.float32)
  return convolution_cython(input, Gy)
```

Listing 3: Versión simple de Listing 12 y 13

Cabe destacar que la función utilizada para calcular la convolución es `convolution_cython` la cual no será explicada en detalle, puesto que no es parte del objetivo de la tarea, pero que sí se incluye en el Anexo (**Listing 11**). Y también cabe destacar que estas funciones se programan en `cython`, las versiones mostradas en el listing anterior son equivalentes pero se presentan así en favor de la claridad.

3.4. Matrices de magnitud y orientación del gradiente

Como se explicó en la **Sección 2.1**, una vez obtenidos los gradientes en `x` y en `y`, el siguiente paso es calcular la magnitud y la orientación del gradiente en cada posición `[y,x]` de la imagen.

La función encargada de calcular la magnitud del gradiente es `magMatrix(gx, gy)`, que recibe como entradas los gradientes en x (`gx`) y en y (gy). Esta función básicamente recorre ambas matrices pixel a pixel, asumiendo que son del mismo tamaño, y para cada posición `[y,x]` calcula la magnitud del gradiente mediante:

$$mag(y,x) = \sqrt{G_x(y,x)^2 + G_y(y,x)^2}$$

lo que en el código se traduce a:

```
def magMatrix(gx, gy):
  out = np.zeros(gx.shape, np.float32)
  for y in range(gx.shape[0]):
    for x in range(gy.shape[1]):
      out[y,x] = np.sqrt(np.square(gx[y,x]) + np.square(gy[y,x]))
  return out
```

Listing 4: Versión simple de Listing 14

La función que se encarga de calcular la orientación del gradiente es `angMatrix(gx, gy)`, que recibe como entradas los gradientes en x (`gx`) y en y (`gy`). Esta función, al igual que `magMatrix` simplemente recorre ambas matrices pixel a pixel y para cada posición `[y,x]` calcula la orientación del gradiente mediante:

$$ang(y,x) = \begin{cases} arctan(\frac{G_y(y,x)}{G_x(y,x)}) & si \;\; G_x > 0, \;\; G_y > 0 \\ \frac{\pi}{2} & si \;\; G_x = 0, \;\; G_y > 0 \\ -\frac{\pi}{2} & si \;\; G_x = 0, \;\; G_y < 0 \\ \text{indefinido} & si \;\; G_x = 0, \;\; G_y = 0 \end{cases}$$

Cabe destacar que los valores que se utilizan para HOG deben estar en grados, y los entregados por esta función se encuentran en radianes, por lo que es necesario transformarlos. Para esto basta con utilizar la función `rad2deg()` de `numpy`.

Dado que la función $ang(y,x)$ tiene como recorrido $[-\frac{\pi}{2}, \frac{\pi}{2}]$, se deben escalar sus salidas para que estas estén en $[0, \pi]$, para lo cual se utiliza la función `minMaxScaler(array, min, max)`, que recibe como entradas el array que se quiere transformar (`array`), el valor mínimo deseado (`min`) y el valor máximo deseado (`max`).

Para transformar el array `minMaxScaler` simplemente aplica la siguiente función:

$$array' = (\frac{array - array.min()}{array.max() - array.min()}) \cdot (max - min) + min \tag{2}$$

Por lo tanto la función `angMatrix` aplica las ideas recién descritas, lo que en el código se traduce a :

```
def minMaxScaler(array, minV, maxV):
  std = (array - array.min()) / (array.max() - array.min())
  return std * (maxV - minV) + minV

def angMatrix(gx, gy):
  out = np.zeros(gx.shape, np.float32)
  for y in range(gx.shape[0]):
    for x in range(gy.shape[1]):
      if gx[y,x] == 0.0:
        if gy[y,x] > 0:
          out[y,x] = 90.0
        if gy[y,x] < 0:
          out[y,x] = -90.0
      else:
        out[y,x] = np.rad2deg(np.arctan(gy[y,x]/gx[y,x]))
  return minMaxScaler(out, 0.001, 179.999)
```

Listing 5: Versión simple de Listing 15 y 16

3.5. Histograma de gradiente orientado

Como se explicó en la **Sección 2.1**, luego de calcular las matrices de magnitud y orientación del gradiente, sólo falta dividir la imagen en 8 x 16 celdas de (8,8) y calcular el histograma del gradiente. La función encargada de esto es `computeHOG(img)`, que recibe como entrada la imagen para la cual se quieren calcular las características HOG (`img`). Para dividir la imagen en las celdas basta con recorrer la imagen usando dos for loops y tomando `slices` siguiendo los múltiplos de 8. Como también se explicó en la Sección 2.1, el histograma de gradientes se construye registrando los votos según:

$$bin[\lfloor \frac{\theta}{20^\circ} \rfloor] \stackrel{+}{=} \mu \cdot dist((y,x),(y_c,x_c))$$

La función `computeHOG()` realiza este cálculo mediante la función auxiliar `getHistogram(magM, angM)`, que recibe como entradas slices de tamaño (8,8) de las matrices de magnitud (`magM`) y de orientación (`angM`), la cual básicamente recorre ambas matrices, pixel a pixel, y para cada posición aplica la fórmula anterior:

```
def getHistogram(magM, angM):
  bins = np.zeros(9, np.float32)
  for y in range(angM.shape[0]):
    for x in range( angM.shape[1]):
        bins[int(np.floor(angM[y,x]/20))] += magM[y,x]*normDist(dist((y,x), (3,3)))
  return bins
```

Listing 6: Versión simple de Listing 19

En esta función se pondera la magnitud del gradiente por la distancia de la posición `[y,x]` con respecto al centro de la ventana de (8,8) (que se escoge de manera arbitraria como `[3,3]`), lo cual se realiza mediante la función auxiliar `dist(coord1, coord2)` que recibe un par de tuplas que contienen las coordenadas (`coord1`, `coord2`).

Pero si sólo se ponderase por la distancia, posiciones más alejadas del centro tendrían votos más importantes, por lo que se normaliza la distancia de modo que los valores más lejanos sean 0 y los más cercanos 1, lo cual se hace mediante la función auxiliar `normDist(dis)` que recibe como entrada la distancia calculada entre dos coordenadas dentro de la ventana de (8,8) (`dis`).

Las funciones recién mencionadas simplemente calculan una distancia euclidiana y aplican la misma fórmula del minMaxScaler, pero dejando como valor mínimo 0.0001, para evitar problemas de posibles divisiones por cero. Estas funciones en el código se traducen a:

```
def dist(coord1, coord2):
  return np.sqrt((coord2[0] - coord1[0])**2 + (coord2[1] - coord1[1])**2)

def normDist(dis):
  return 0.0001 + (dis - dist((0,0),(4,4)))/(0 - dist((0,0),(4,4)))
```

Listing 7: Versión simple de Listing 17 y 18

Finalmente, combinando todas las funciones recién descritas, y la forma de obtener las ventanas de (8,8), se construye la función `computeHOG(img)`, la cuale en el código se traduce a:

```
def computeHOG(img):
  HOG = np.zeros((16,8,9), np.float32)
  # gradients
  imgGradX = gradx(img)
  imgGradY = grady(img)
  # magnitude and orientation
  magnitude = magMatrix(imgGradX, imgGradY)
  angle = angMatrix(imgGradX, imgGradY)
  # histograms
  for y in range(16):
    for x in range(8):
      HOG[y,x,:] = getHistogram(magnitude[8*y:8*(y+1),8*x:8*(x+1)],angle[8*y:8*(y
    +1),8*x:8*(x+1)])
  return HOG
```

Listing 8: Versión simple de Listing 20

El último paso para formar el vector de características HOG, como también se menciona en la Sección 2.1, consiste en formar bloques de 4 celdas, concatenar sus histogramas, normalizar el nuevo histograma de tamaño 36, y concatenar todos los histogramas resultantes de ese procedimiento. La función encargada de esto es `blockNorm(HOG)`, que recibe como entrada las características HOG sin normalizar (`HOG`), y que en el código se traduce a:

```
def blockNorm(HOG):
  vec = np.array([], np.float32)
  for y in range(HOG.shape[0] - 1):
    for x in range(HOG.shape[1] - 1):
      current = np.concatenate((HOG[y,x], HOG[y,x+1],
                                HOG[y+1,x], HOG[y+1,x+1]), axis = 0)
      norm = current/np.sum(current)
      vec = np.concatenate((vec,norm), axis = 0)
  return vec
```

Listing 9: Versión simple de Listing 21

3.6. Extraer características de los conjuntos

Ya teniendo las funciones necesarias para calcular las características HOG normalizadas se deben calcular las características de todas las imágenes entregadas por el cuerpo docente, con el fin de entrenar los clasificadores.

Para esto basta con, para cada carpeta de imágenes, iterar sobre las imágenes que contiene, convertirlas a escala de grises y re-dimensionarlas, calcular sus características HOG y normalizarlas:

```
# conjunto de entrenamiento
HOG_X_train = np.zeros((X_train.shape[0],3780))
for i in range(len(X_train)):
  img = cv2.imread(f"{PATH}/{X_train[i]}", 0)
  img = grayAndResize(img)
  HOG = computeHOG(img)
  HOG_X_train[i,:] = blockNorm(HOG)

# conjunto de validacion
HOG_X_val = np.zeros((X_val.shape[0],3780))
for i in range(len(X_val)):
  img = cv2.imread(f"{PATH}/{X_val[i]}", 0)
  img = grayAndResize(img)
  HOG = computeHOG(img)
  HOG_X_val[i,:] = blockNorm(HOG)

# conjunto de prueba
HOG_X_test = np.zeros((X_test.shape[0],3780))
for i in range(len(X_test)):
  img = cv2.imread(f"{PATH}/{X_test[i]}", 0)
  img = grayAndResize(img)
  HOG = computeHOG(img)
  HOG_X_test[i,:] = blockNorm(HOG)
```

Listing 10: Versión simple de Listing 21

4. Entrenar clasificadores

Debido a que se utilizan los clasificadores de `sklearn`, entrenarlos consiste en básicamente llamar tres funciones, una para crear el clasificador, otra para entrenarlo y otra para predecir, que en el caso de SVM y RF es:

```
clasifSVM = svm.SVC()
clasifSVM.fit(HOG_X_train, y_train)
valPred = clasifSVM.predict(HOG_X_val)

clasifRF = RandomForestClassifier()
clasifRF.fit(HOG_X_train, y_train)
valPred = clasifRF.predict(HOG_X_val)
```

Lo más relevante de este procedimiento es buscar los mejores hiper-parámetros, lo cual se hace mediante `gridSearchCV`, que hace una búsqueda de los hiper-parámetros probando distintas combinaciones y evaluando cuál es la mejor combinación. En esta búsqueda se aplica validación cruzada mediante `predefinedSplit`.

Para usar `predefinedSplit` se debe crear una lista que contenga unos y menos unos, con tantos 1 como elementos de entrenamiento y -1 como elementos de validación:

```
fold = []
for i in HOG_X_train:
  fold.append(1)
for i in HOG_X_val:
  fold.append(-1)

ps = PredefinedSplit(fold)
```

Una vez definido el `predefinedSplit`, se puede hacer la búsqueda de parámetros con `gridSearchCV`, utilizando tanto los datos de entrenamiento como los de validación, que para el caso de la SVM es:

```
all_train_data = np.concatenate((HOG_X_train,HOG_X_val), axis = 0)
all_train_labels = np.concatenate((y_train,y_val), axis = 0)

param_grid = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['linear', 'poly', 'rbf', 'sigmoid']}

grid = GridSearchCV(svm.SVC(), param_grid, refit = True, cv = ps)
grid.fit(all_train_data, all_train_labels)
```

Y para el caso de RF es:

```
param_grid = {'n_estimators': [10, 100, 500, 1000],
              'criterion': ['gini', 'entropy'],
              'max_depth': [1, 10, 50, 100, 500, None]}

grid = GridSearchCV(RandomForestClassifier(), param_grid, refit = True, cv = ps)
grid.fit(all_train_data, all_train_labels)
```

Los mejores clasificadores encontrados con esta metodología son:

$$\begin{aligned} &SVM(C = 10, gamma = 0{,}1, kernel = \text{'rbf'}) \\ &RF(criterion = \text{'entropy'}, max\_depth = 100, n\_estimators = 100) \end{aligned} \tag{1}$$

5. Resultados

5.1. Clasificación SVM

Al entrenar el clasificador SVM con los hiper-parámetros predeterminados y predecir sobre el conjunto de validación se obtiene una `accuracy` de 0.61, y una precisión, recall y f-1 score de:

Tabla 1: Precisión, recall y f1-score para SVM base en validación

clase	precision	recall	f1-score
0	0.94	0.57	0.71
1	0.36	0.63	0.46
2	0.69	0.65	0.67

La matriz de confusión de esta predicción se muestra en la **Figura 10**:

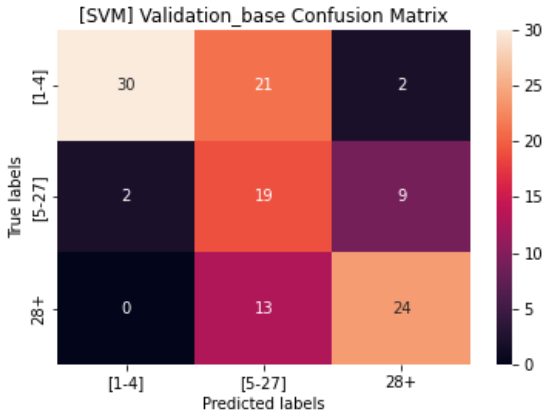

Figura 10: Matriz de confusión SVM base en conjunto de validación

Al entrenar la SVM con los mejores hiper-parámetros y predecir sobre el conjunto de validación se obtiene una `accuracy` de 0.71, y una precisión, recall y f-1 score de:

Tabla 2: Precisión, recall y f1-score para SVM en validación

clase	precision	recall	f1-score
0	0.95	0.72	0.82
1	0.45	0.63	0.53
2	0.74	0.76	0.75

La matriz de confusión de esta predicción se muestra en la **Figura 11**:

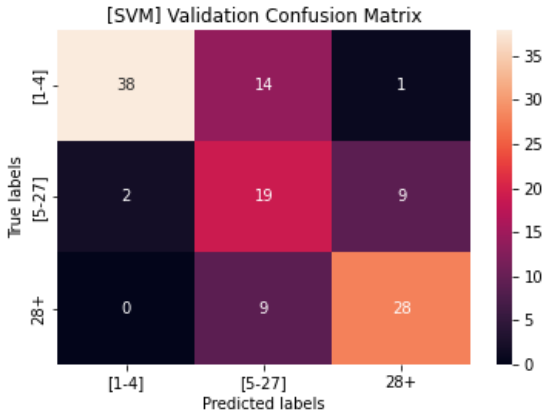

Figura 11: Matriz de confusión SVM en conjunto de validación

Y al predecir sobre el conjunto de prueba se obtiene una `accuracy` de 0.84, y una precisión, recall y f-1 score de:

Tabla 3: Precisión, recall y f1-score para SVM en prueba

clase	precision	recall	f1-score
0	0.93	0.90	0.91
1	0.68	0.81	0.74
2	0.90	0.81	0.85

La matriz de confusión de esta predicción se muestra en la **Figura 12**:

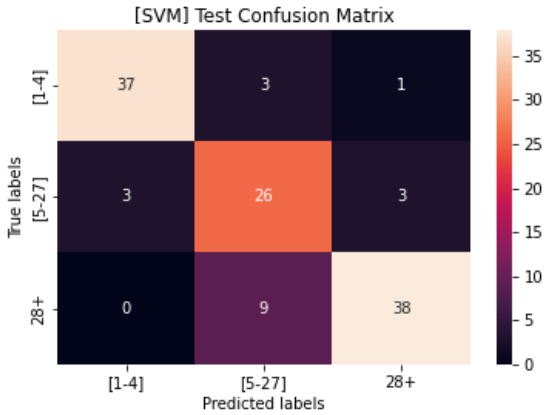

Figura 12: Matriz de confusión SVM en conjunto de prueba

5.2. Clasificación RF

Al entrenar el clasificador RF con los hiper-parámetros predeterminados y predecir sobre el conjunto de validación se obtiene una `accuracy` de 0.68, y una precisión, recall y f-1 score de:

Tabla 4: Precisión, recall y f1-score para RF base en validación

clase	precision	recall	f1-score
0	0.95	0.68	0.79
1	0.42	0.63	0.51
2	0.73	0.73	0.73

La matriz de confusión de esta predicción se muestra en la **Figura 13**:

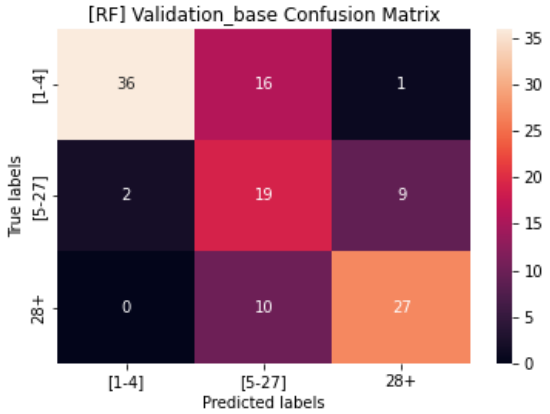

Figura 13: Matriz de confusión RF base en conjunto de validación

Al entrenar RF con los mejores hiper-parámetros y predecir sobre el conjunto de validación se obtiene una `accuracy` de 0.66, y una precisión, recall y f-1 score de:

Tabla 5: Precisión, recall y f1-score para RF en validación

clase	precision	recall	f1-score
0	0.94	0.64	0.76
1	0.41	0.70	0.52
2	0.73	0.65	0.69

La matriz de confusión de esta predicción se muestra en la **Figura 14**:

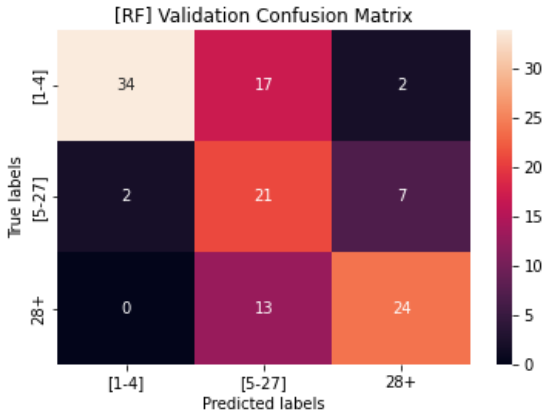

Figura 14: Matriz de confusión RF en conjunto de validación

Y al predecir sobre el conjunto de prueba se obtiene una `accuracy` de 0.69, y una precisión, recall y f-1 score de:

Tabla 6: Precisión, recall y f1-score para RF en prueba

clase	precision	recall	f1-score
0	0.82	0.68	0.75
1	0.47	0.78	0.59
2	0.91	0.64	0.75

La matriz de confusión de esta predicción se muestra en la **Figura 15**:

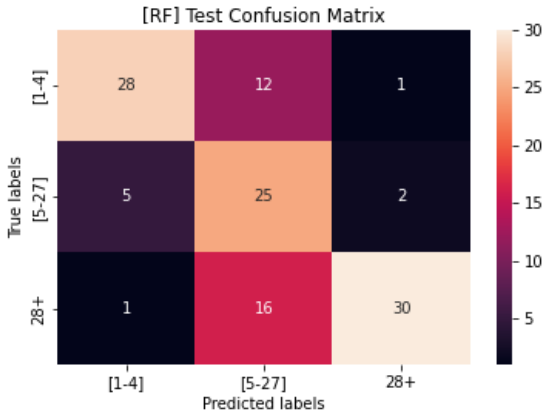

Figura 15: Matriz de confusión RF en conjunto de prueba

6. Análisis de resultados

Se puede afirmar que las características HOG implementadas son suficientes para resolver el problema, puesto que ambos clasificadores tuvieron un desempeño bueno.

6.1. Clasificación SVM

El mejor clasificador SVM logra resultados destacables, con una `accuracy` del 0.84 que implica muy pocas clasificaciones erróneas, como se puede apreciar en la **Figura 12**, donde la mayor cantidad de errores se obtuvieron al predecir que individuos mayores a 28 años tenían entre 5 y 7.

Se puede apreciar que la búsqueda de los mejores hiper-parámetros vale completamente la pena, basta con comparar las **Figuras 10 y 11** para darse cuenta de que la mejor versión tiene muchas menos clasificaciones erróneas, afirmación respaldada por las métricas presentadas en las **Tablas 1 y 2**, donde se puede apreciar que el f1-score de la mejor versión es superior en todas las clases a la versión base.

6.2. Clasificación RF

Se puede decir básicamente lo mismo de RF. El mejor clasificador alcanza un buen resultado con una `accuracy` de 0.69 y con no muchas clasificaciones erróneas, como se puede ver en la **Figura 15**, donde la mayor confusión sigue siendo clasificar individuos mayores a 28 como personas entre 5 a 27, pero aquí también hay una cantidad considerable de individuos entre 1 y 4 años que también son clasificados entre 5 a 27.

En este caso, buscar los mejores hiper-parámetros mejora el clasificador un poco en la clase 1 ([5,27]), pero empeora su desempeño en las clases 0 ([1,4]) y 2 (28+). Esto puede deberse a que la búsqueda de hiper-parámetros no fue lo suficientemente extensa, es decir, se debieron haber incluido más opciones para los hiper-parámetros en `gridSearchCV`.

Se puede afirmar que el clasificador SVM es mejor que el RF en la tarea de clasificación de edad, puesto que SVM obtuvo resultados consistentemente mejores que RF en todas las predicciones. Los clasificadores HOG sí son apropiados para resolver este problema, puesto que logran describir personas con tal precisión que incluso las versiones base de los clasificadores obtienen buenos resultados.

Se pueden realizar dos cosas para mejorar los resultados: En primer lugar, mejorar las características HOG, para lo cual se podría hacer que cada posición `[y,x]` del gradiente vote en múltiples orientaciones, ponderando los votos además por qué tan cerca se encuentra la orientación de los respectivos bines. Otra forma de mejorar los resultados sería mejorando los clasificadores, para lo cual se podrían buscar mejores hiper-parámetros haciendo búsquedas más intensivas, o se podría probar con otros clasificadores (como redes neuronales) para ver si se obtiene un mejor desempeño.

7. Conclusión

Se logra implementar en Python todas las funciones necesarias tanto para el cálculo de las características HOG, como para el entrenamiento y predicción de los clasificadores. Queda de manifiesto la eficiencia de estas características en la tarea de clasificar rostros humanos, así como la importancia y utilidad de buscar los hiper-parámetros óptimos de los clasificadores. También se concluye que, en la tarea de clasificar rostros humanos, las SVM son superiores a los RF.

Debido a que la mayoría de las funciones se programaron en bajo nivel, es decir, utilizando casi exclusivamente las funciones básicas de python y numpy, se logra adquirir un mayor entendimiento sobre el funcionamiento de estos métodos y sobre las dificultades de implementarlos.

8. Anexo

8.1. Extracción de características HOG

```
%%cython
import numpy as np
import math
cimport numpy as np

#@cython.boundscheck(False)
cpdef np.ndarray[np.float32_t, ndim=2] convolution_cython(np.ndarray[np.float32_t,
    ndim=2] input, np.ndarray[np.float32_t, ndim=2] mask):
  cdef int y, x, rows, cols, kernelRows, kernelCols, offset
  cdef float sum, pixel
  cdef np.ndarray[np.float32_t, ndim=2] output = np.zeros([input.shape[0],input.
   shape[1]], dtype = np.float32)

  # tamano de la imagen
  rows = input.shape[0]
  cols = input.shape[1]

  # tamanno del kernel
  kernelRows = mask.shape[0]
  kernelCols = mask.shape[1]

  # como encontrar los pixeles correspondientes
  offsetX = math.floor(kernelRows/2)
  offsetY = math.floor(kernelCols/2)

  # convolucion
  for y in range(rows):
    for x in range(cols):
      sum = 0
      for kernelY in range(kernelRows):
        for kernelX in range(kernelCols):
          posX = (x - offsetX) + kernelX
          posY = (y - offsetY) + kernelY
          if ((min(posX,posY) < 0) | (posX >= cols) | (posY >= rows)):
            pixel = 0.0
          else:
            pixel = input[posY,posX]
          sum += pixel*mask[kernelY,kernelX]
      output[y,x] = sum
  return output
```

Listing 11: convolution_cython

```
cpdef np.ndarray[np.float32_t, ndim=2] gradx(np.ndarray[np.float32_t, ndim=2] input
    ):
  cdef np.ndarray[np.float32_t, ndim = 2] output = np.zeros([input.shape[0], input.
    shape[1]], dtype = np.float32)
  cdef np.ndarray[np.float32_t, ndim = 2] Gx = np.array([[1, 0 ,-1],
                                                         [2, 0, -2],
                                                         [1, 0 , -1]], dtype =
    np.float32)
  output = convolution_cython(input, Gx)
  return output
```

Listing 12: gradx

```
cpdef np.ndarray[np.float32_t, ndim=2] grady(np.ndarray[np.float32_t, ndim=2] input
    ):
  cdef np.ndarray[np.float32_t, ndim=2] output=np.zeros([input.shape[0], input.
    shape[1]], dtype = np.float32)
  cdef np.ndarray[np.float32_t, ndim = 2] Gy = np.array([[1, 2 ,1],
                                                         [0, 0, 0],
                                                         [-1, -2 , -1]], dtype
    = np.float32)

  output = convolution_cython(input, Gy)
  return output
```

Listing 13: grady

```
cpdef np.ndarray[np.float32_t, ndim=2] magMatrix(np.ndarray[np.float32_t, ndim=2]
    gx, np.ndarray[np.float32_t, ndim=2] gy):
  cdef np.ndarray[np.float32_t, ndim=2] out
  cdef int x, y, shapeX, shapeY
  shapeY = gx.shape[0]
  shapeX = gy.shape[1]
  out = np.zeros((shapeY, shapeX), np.float32)
  for y in range(shapeY):
    for x in range(shapeX):
      out[y,x] = np.sqrt(np.square(gx[y,x]) + np.square(gy[y,x]))
  return out
```

Listing 14: magMatrix

```
cpdef np.ndarray[np.float32_t, ndim=2] minMaxScaler(np.ndarray[np.float32_t, ndim
    =2] array, float min, float max):
  cdef np.ndarray[np.float32_t, ndim=2] std
  std = (array - array.min()) / (array.max() - array.min())
  return std * (max - min) + min
```

Listing 15: minMaxScaler

```
cpdef np.ndarray[np.float32_t, ndim=2] angMatrix(np.ndarray[np.float32_t, ndim=2]
    gx, np.ndarray[np.float32_t, ndim=2] gy):
  cdef np.ndarray[np.float32_t, ndim=2] out
  cdef int x, y, shapeX, shapeY
  shapeY = gx.shape[0]
  shapeX = gy.shape[1]
  out = np.zeros((shapeY, shapeX), np.float32)
  for y in range(shapeY):
    for x in range(shapeX):
      # para no dividir por cero
      if gx[y,x] == 0.0:
        if gy[y,x] > 0:
          out[y,x] = 90.0
        if gy[y,x] < 0:
          out[y,x] = -90.0
      else:
        out[y,x] = np.rad2deg(np.arctan(gy[y,x]/gx[y,x]))
  # out vive en [-90 a 90], la llevamos a [0,180]
  out = minMaxScaler(out, 0.001, 179.999)
  return out
```

Listing 16: angMatrix

```
cpdef float dist(tuple coord1, tuple coord2):
  return np.sqrt((coord2[0] - coord1[0])**2 + (coord2[1] - coord1[1])**2)
```

Listing 17: dist

```
cpdef float normDist(float dis):
  return 0.0001 + (dis - dist((0,0),(4,4)))/(0 - dist((0,0),(4,4)))
```

Listing 18: normDist

```
cpdef np.ndarray[np.float32_t, ndim=1] getHistogram(np.ndarray[np.float32_t, ndim
    =2] magM, np.ndarray[np.float32_t, ndim=2] angM):
  cdef np.ndarray[np.float32_t, ndim=1] bins = np.zeros(9, np.float32)
  cdef int x, y, shapeX, shapeY
  shapeY = angM.shape[0]
  shapeX = angM.shape[1]
  for y in range(shapeY):
    for x in range(shapeX):
        bins[int(np.floor(angM[y,x]/20))] += magM[y,x]*normDist(dist((y,x), (3,3)))
  return bins
```

Listing 19: getHistogram

```
cpdef np.ndarray[np.float32_t, ndim=3] computeHOG(np.ndarray[np.float32_t, ndim=2] img):
  cdef np.ndarray[np.float32_t, ndim=3] HOG = np.zeros((16,8,9), np.float32)
  cdef np.ndarray[np.float32_t, ndim=2] imgGradX, imgGradY, magnitude, angle
  cdef int x, y
  # gradients
  imgGradX = gradx(img)
  imgGradY = grady(img)
  # magnitude and orientation
  magnitude = magMatrix(imgGradX, imgGradY)
  angle = angMatrix(imgGradX, imgGradY)
  # histograms
  for y in range(16):
    for x in range(8):
      HOG[y,x,:] = getHistogram(magnitude[8*y:8*(y+1),8*x:8*(x+1)],angle[8*y:8*(y+1),8*x:8*(x+1)])
  return HOG
```

Listing 20: computeHOG

```
cpdef  np.ndarray[np.float32_t, ndim=1] blockNorm(np.ndarray[np.float32_t, ndim=3] HOG):
  cdef np.ndarray[np.float32_t, ndim=1] vec = np.array([], np.float32)
  cdef np.ndarray[np.float32_t, ndim=1] current
  cdef np.ndarray[np.float32_t, ndim=1] norm
  cdef int x, y, shapeX, shapeY
  shapeX = HOG.shape[1] - 1
  shapeY = HOG.shape[0] - 1
  for y in range(shapeY):
    for x in range(shapeX):
      current = np.concatenate((HOG[y,x], HOG[y,x+1], HOG[y+1,x], HOG[y+1,x+1]), axis = 0)
      norm = current/np.sum(current)
      if np.isnan(norm).any():
      vec = np.concatenate((vec,norm), axis = 0)
  return vec
```

Listing 21: blockNorm

8.2. Entrenar clasificadores

```
def drawConfusionMatrix(y_true, y_pred, title):
  cm = confusion_matrix(y_true, y_pred)
  ax= plt.subplot()
  sns.heatmap(cm, annot=True, fmt='g', ax=ax)
  ax.set_xlabel('Predicted labels')
  ax.set_ylabel('True labels')
  ax.set_title(f'{title} Confusion Matrix')
  ax.xaxis.set_ticklabels(['[1-4]', '[5-27]', '28+'])
  ax.yaxis.set_ticklabels(['[1-4]', '[5-27]', '28+'])
  plt.savefig(f'{title}_CM.png')
```

Listing 22: drawConfusionMatrix

```
clasifSVM = svm.SVC()
clasifSVM.fit(HOG_X_train, y_train)
valPred = clasifSVM.predict(HOG_X_val)
print(classification_report(y_val, valPred))
drawConfusionMatrix(y_val, valPred, "[SVM] Validation_base")
```

```
fold = []
for i in HOG_X_train:
  fold.append(1)
for i in HOG_X_val:
  fold.append(-1)

ps = PredefinedSplit(fold)
all_train_data = np.concatenate((HOG_X_train,HOG_X_val), axis = 0)
all_train_labels = np.concatenate((y_train,y_val), axis = 0)
```

```
param_grid = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['linear', 'poly', 'rbf', 'sigmoid']}

grid = GridSearchCV(svm.SVC(), param_grid, refit = True, cv = ps)

grid.fit(all_train_data, all_train_labels)
print(grid.best_params_)
```

```
clasifSVM = svm.SVC(**grid.best_params_)
clasifSVM.fit(HOG_X_train, y_train)

valPred = clasifSVM.predict(HOG_X_val)
print(classification_report(y_val, valPred))
drawConfusionMatrix(y_val, valPred, "[SVM] Validation")
```

```
predictions = clasifSVM.predict(HOG_X_test)
print(classification_report(y_test, predictions))
drawConfusionMatrix(y_test, predictions, "[SVM] Test")
```

```
clasifRF = RandomForestClassifier()
clasifRF.fit(HOG_X_train, y_train)
valPred = clasifRF.predict(HOG_X_val)
print(classification_report(y_val, valPred))
drawConfusionMatrix(y_val, valPred, "[RF] Validation_base")
```

```
param_grid = {'n_estimators': [10, 100, 500, 1000],
              'criterion': ['gini', 'entropy'],
              'max_depth': [1, 10, 50, 100, 500, None]}

grid = GridSearchCV(RandomForestClassifier(), param_grid, refit = True, cv = ps)

grid.fit(all_train_data, all_train_labels)
print(grid.best_params_)
```

```
clasifRF = RandomForestClassifier(**grid.best_params_)
clasifRF.fit(HOG_X_train, y_train)

valPred = clasifRF.predict(HOG_X_val)
```

```
print(classification_report(y_val, valPred))
drawConfusionMatrix(y_val, valPred, "[RF] Validation")
```

```
predictions = clasifRF.predict(HOG_X_test)
print(classification_report(y_test, predictions))
drawConfusionMatrix(y_test, predictions, "[RF] Test")
```

Referencias

[1] Apunte de las clases de Procesamiento Avanzado de Imágenes - EL7008 - 1, Primavera 2021, Javier Ruiz del Solar.

[2] Apunte de las clases de Inteligencia Computacional - EL4106-1, Primavera 2020, Pablo Estevez V.

[3] Tarea 3 - EL7008 (Primavera 2021): Pirámides de Gauss y Laplace, Javier Ruiz del Solar, Patricio Loncomilla.

[4] "Histogram of oriented gradients," Wikipedia, 01-Feb-2021. [Online]. Disponible en: https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients. [Acceso: 13-Oct-2021].