Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Tarea 4

Detección de Personas usando Adaboost y características tipo Haar

Integrantes: Benjamín A. Irarrázabal T.
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Ayudantes: José Díaz V.
Danilo Moreira
Javier Mosnaim Z.
Jhon Pilataxi

Fecha de realización: 23 de octubre de 2022
Fecha de entrega: 23 de octubre de 2022
Santiago de Chile

Índice de Contenidos

Índice de Figuras

1. Introducción

El procesamiento de imágenes se compone por un conjunto de técnicas aplicadas a imágenes digitales para poder modificar su calidad o buscar información dentro de esta, por ejemplo, colores, bordes, entre otros. Estos métodos han tenido un gran impacto en variadas áreas tales como medicina, telecomunicaciones, industria e incluso entretenimiento. Dentro de estos procedimientos, se encuentra la detección de personas (o de cualquier otro objeto en particular) mediante el entrenamiento de diferentes algoritmos de clasificación. Para esto, se hace enfoque a dos procesos importantes, en primer lugar la extracción de características de una imagen, la cual puede realizarse con métodos tales como HOG, LBP, redes convolucionales, Haar, entre otros. Luego, con estas características (*features*), se procede a realizar la clasificación. Lo anterior puede realizarse en base a algoritmos de aprendizaje supervisado y no supervisado. En esta experiencia en particular, se trabajará con *Haar Features* y clasificación mediante *Adaboost*.

En base a lo anterior, el presente informe tiene como objetivo el desarrollo de la cuarta tarea del curso Procesamiento Avanzado de Imágenes y busca, utilizando herramientas computacionales y programación en lenguaje Python, la extracción de características mediante *Haar* y la posterior clasificación usando *Adaboost*, ambos implementados desde cero, sobre imágenes como las presentadas en la sección 6.1. Para esto, se comenzará entregando una breve introducción de los algoritmos utilizados junto a su código implementado, luego se presentarán los resultados obtenidos para las clasificaciones con su respectivo análisis, para finalmente, entregar las principales conclusiones de la experiencia.

2. Algoritmos

2.1. Haar Features

Para el cálculo de características Haar se deben destacar los siguientes pilares fundamentales. En primer lugar, es importante mostrar visualmente las máscaras rectangulares utilizadas en la tarea. Estas se pueden observar a continuación.

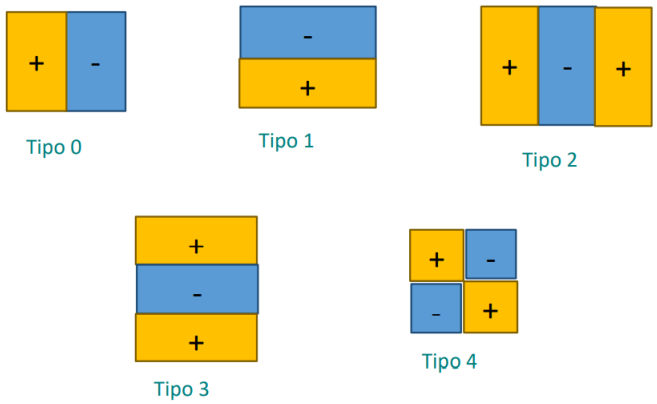


Figura 1: Máscaras rectangulares para obtener características Haar

No obstante, antes de aplicar estas máscaras a las imágenes, se le deben realizar algunas "modificaciones" a estas. Primero, se transforma la imagen a escala de grises y se dimensiona a 24x24 píxeles usando la función **redimension()** de la parte 1 (6.3.1). Luego, se calcula la imagen integral de cada una. Esta, consiste en una suma acumulada de la imagen a partir de los píxeles anteriores, formalmente, suponiendo que I es nuestra imagen (matriz de 24x24) e i,j es una posición arbitraria en esta, el valor de la imagen integral II en la posición i,j se calcularía de la siguiente manera (raster scanning) [2].

$$II_{i,j} = \sum_{m=0}^{i} \sum_{n=0}^{j} I_{m,n} \tag{1}$$

Este proceso se resume en la función **Integral_Image()**, la cual realiza iterativamente esta suma, analizando los casos bordes debidamente (ver sección 6.3.2).
Posterior a esto, se tiene la función **Haar_Mask()** que parametriza las máscaras según su posición, tipo y polaridad, esta última define los signos de la figura 1, si la polaridad es -1, los signos se invierten, por otro lado, si es 1, se mantienen (sección 6.3.3 del código).
Finalmente, la extracción de características se genera gracias a la función **Haar_Features**, la cual recibe el conjunto de imágenes y la lista de parámetros y realiza la operación asociada a esta. Estas operaciones fueron revisadas de manera externa a mano por el autor, este procedimiento se puede observar en la sección 6.2 del anexo, donde se realiza el análisis según las esquinas con coordenadas $(x1, y1)$ y $(x2, y2)$. (ver sección 6.3.4)

2.2. Clasificador Adaboost

El entrenamiento y la clasificación del algoritmo Adaboost, se encuentra en la sección 6.3.5, en esta se observan 6 funciones importantes, las cuales se detallan a continuación.

- **h:** esta función recibe dos parámetros, x (arreglo de números) y u (un umbral). Su acción es sencilla, utiliza los signos > y <= (mayor y menor o igual) para comprobar los números de cada posición en x y el umbral definido. Si la comparación se cumple, se retorna un valor booleano true en esa posición, por el contrario se retorna false. Esto es útil ya que posterior a esto, se puede multiplicar este nuevo arreglo booleano por 1 o -1 según corresponda para posteriormente sumar ambos y cumplir el propósito, creando la base de un clasificador débil.

- **Best_U:** como dice su nombre, esta función calcula los mejores umbrales (y también los máximos valores de r) a partir de X (una matriz de características), y (el vector de clases) y w (vector de pesos). Con esto, la función recorre las columnas de la matriz X (es decir, cada característica) y toma su valor máximo y su valor mínimo, para posteriormente, realizar un arreglo equidistante para u con diez datos entre estos mencionados. Con estos umbrales, realiza el cálculo de r con la siguiente ecuación:

$$r = \sum_{k=0}^{N-1} w_k \cdot y_k \cdot h(x_{k,i}, u) \tag{2}$$

  Luego de esta operación, buscamos el mayor r (max(r)) y su índice correspondiente para guardar el mejor umbral. Con esto, se tendrá un arreglo para los mejores umbrales y su respectivo r asociado.

- **a_t:** esta función, realiza el cálculo de alpha en base a un r entregado usando la siguiente ecuación.

$$\alpha = 0.5 \cdot \ln\left(\frac{1+r}{1-r}\right) \tag{3}$$

- **Train_Adaboost:** una de las más importantes, es la encargada del entrenamiento del algoritmo de clasificación. En esta, se inicializan los pesos como $\frac{1}{N}$ (con N cantidad de ejemplos) y se itera T veces (con el fin de seleccionar T clasificadores débiles). En cada una de estas, se calculan los mejores umbrales y r con la función Best_U y se calculan los alphas correspondientes con a_t, luego de esto, se busca el r más grande encontrado y su indice asociado. Para cada iteración, se guarda en un arreglo el r mencionado, su índice, el umbral y alpha respectivo. Luego de esto (antes de pasar a la siguiente iteración, se actualizan los pesos mediante la siguiente ecuación (luego se normalizan).

$$w_{t+1} = w_t \cdot \exp(-\alpha_t \cdot y \cdot h_t(x, u_t)) \tag{4}$$

- **Adaboost_classify:** utiliza lo retornado en Train_Adaboost, es decir, at, ut e it (alphas, umbrales e índices) y realiza la clasificación mediante la creación de un clasificador fuerte a partir de los T débiles en base a la siguiente ecuación.

$$H(x) = signo\left(\sum_{t=0}^{T-1} \alpha_t \cdot h_t(x)\right) \tag{5}$$

  La cual para cada imagen, entregará un 1 o un -1 dependiendo de su clasificación persona o no persona respectivamente.

- **Classify_all:** esta función toma la anterior y permite aplicarla a una una lista de imágenes de forma iterativa, simplemente utiliza un ciclo for para realizarlo.

2.3. Dibujo de Máscaras

Para dibujar las 5 mejores máscaras obtenidas, se utilizaron las funciones **redimension2** y **Draw_Mask** de la sección 6.3.8, la primera de estas, realiza el dimensionamiento de la imagen a 192x192 píxeles, es decir, que comparando con la imagen que se tenía, se amplifica esta por 8 tanto para su alto como para su ancho. Luego, la función Draw_Mask, utiliza esto, y la el formato de la lista de parámetros entregada por Haar_Mask, para realizar un procedimiento similar al de la extracción de características para analizar las esquinas y píxeles importantes para dibujar los rectángulos. Estos, se dibujan de color rojo, para la polaridad positiva y azules para la negativa. (ver figuras 8, 9 y 10 de la sección 3).

3. Resultados

Para probar los resultados del clasificador Adaboost implementado, se realizaron dos pasos, en primer lugar, se obtiene la matriz de confusión para el conjunto de entrenamiento y el de prueba, y luego, se calcula el accuracy (exactitud) obtenido respectivamente, además, es importante destacar que para estos resultados la semilla utilizada en el parámetro random_state de la función train_test_split fue 20. Estos resultados se presentan a continuación:

3.1. Entrenamiento con T = 10

- Accuracy para el conjunto de entrenamiento: 0.8614864864864865
- Accuracy para el conjunto de prueba: 0.7837837837837838
- Matriz de confusión para el conjunto de entrenamiento:

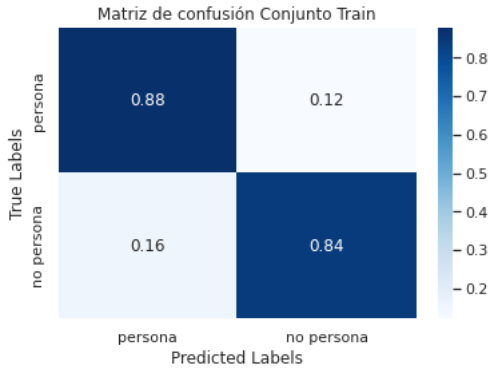


Figura 2: Matriz de confusión para conjunto de entrenamiento usando T = 10

- Matriz de confusión para el conjunto de prueba:

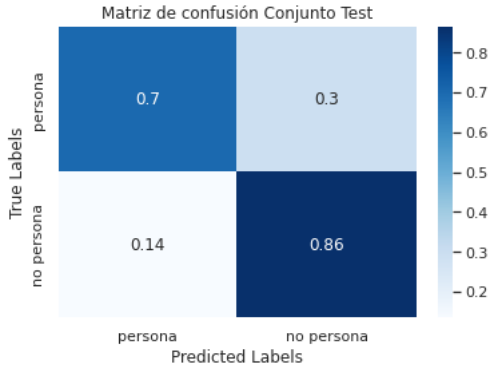


Figura 3: Matriz de confusión para conjunto de prueba usando T = 10

3.2. Entrenamiento con T = 5

- Accuracy para el conjunto de entrenamiento: 0.7905405405405406
- Accuracy para el conjunto de prueba: 0.8378378378378378
- Matriz de confusión para el conjunto de entrenamiento:

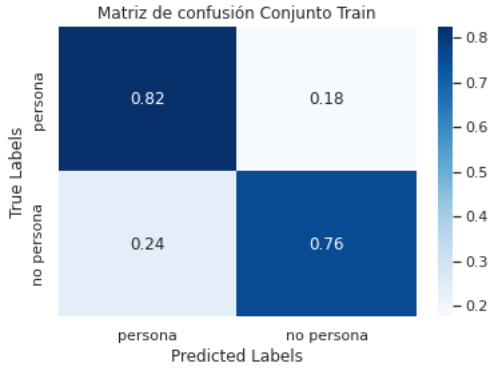


Figura 4: Matriz de confusión para conjunto de entrenamiento usando T = 5

- Matriz de confusión para el conjunto de prueba:

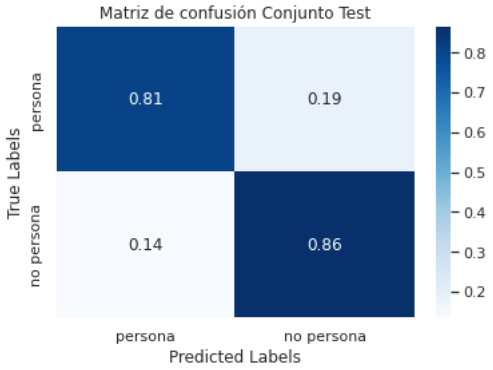


Figura 5: Matriz de confusión para conjunto de prueba usando T = 5

3.3. Entrenamiento con T = 20

- Accuracy para el conjunto de entrenamiento: 0.9391891891891891
- Accuracy para el conjunto de prueba: 0.7297297297297297
- Matriz de confusión para el conjunto de entrenamiento:

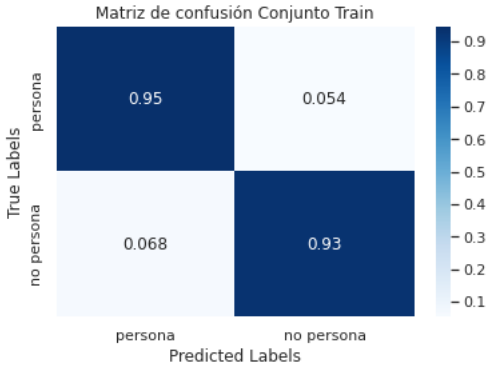


Figura 6: Matriz de confusión para conjunto de entrenamiento usando T = 20

- Matriz de confusión para el conjunto de prueba:

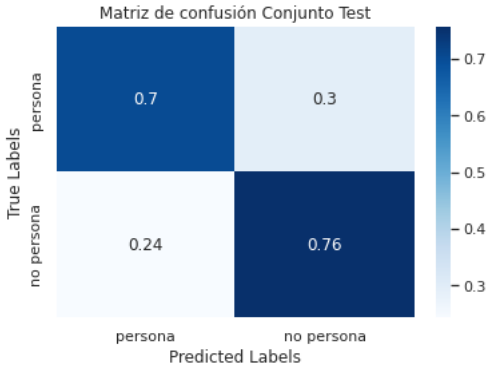


Figura 7: Matriz de confusión para conjunto de prueba usando T = 20

3.4. Dibujo de Máscaras

En las siguientes imágenes y tal como se menciona en la sección 2, las máscaras dibujadas presentan dos colores, rojo para la parte positiva de la máscara y azul para la negativa.

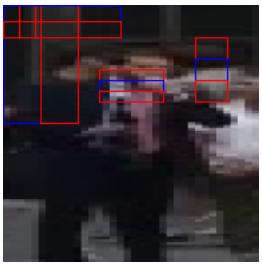


Figura 8: Imagen 1 con máscaras dibujadas

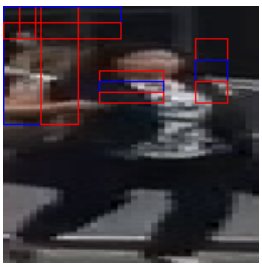


Figura 9: Imagen 2 con máscaras dibujadas

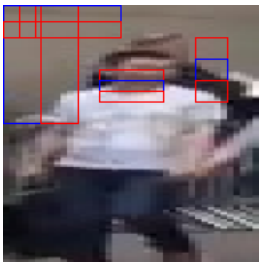


Figura 10: Imagen 3 con máscaras dibujadas

4. Análisis de Resultados

En primer lugar, observando las figuras 2, 4 y 6, correspondientes a las matrices de confusión para el conjunto de entrenamiento en los tres casos probados, se puede destacar que la tercera ($T = 20$) es la que presenta mejores resultados, obteniendo 0.95 y 0.93 en su diagonal, es decir, que acertó correctamente el 95 % de los casos de personas y 93 % para los autos o sillas.
Por otro lado, al observar las figuras 3, 5 y 7, correspondientes a las matrices de confusión para el conjunto de prueba, se puede notar que la segunda ($T = 5$) es la que presenta mejores resultados, obteniendo un 82 % de clasificaciones correctas para personas y 76 % para autos y sillas.

Lo anterior, se puede contrastar con los resultados de la exactitud obtenida en los experimentos, donde el mayor accuracy obtenido para el conjunto de entrenamiento se originó con $T = 20$, alcanzando un 93.92 % aproximadamente, mientras que para el conjunto de prueba se logró utilizando un $T = 5$, obteniendo un 83.78 % aproximadamente.
Otro punto importante a destacar es que con respecto al caso base ($T = 10$), aumentar T mejora la clasificación para el conjunto de entrenamiento, mientras que disminuirlo la empeora, por otro lado, para el conjunto de prueba, aumentar T empeora la clasificación, mientras que disminuirlo la mejora.

Para la sección de dibujo de máscaras, se puede observar una del tipo 0, con polaridad -1, una del tipo 1 con polaridad 1, dos del tipo 3 con polaridad 1 y finalmente una del tipo 4 en la esquina superior izquierda con polaridad 1. Con estas se puede observar que algunas calzan justamente en zonas importantes del objeto en cuestión (persona), como la cara, hombros u otras zonas que permiten identificar y comparar la composición de las imágenes.

Finalmente, es importante mencionar, que se realizaron pruebas con varios valores enteros para el parámetro **random_state**, logrando distintos resultados tantos para el accuracy como para las matrices de confusión, no obstante se escogió 20 ya que permitía entregar un mejor análisis comparativo entre los experimentos variando T.

5. Conclusiones

Observando los resultados de la sección 3, se puede afirmar que se cumplió con el objetivo principal de la experiencia, enfocado en implementar la extracción de características de tipo Haar para la posterior clasificación utilizando el algoritmo Adaboost. Junto a esto, se logró reforzar contenidos vistos en cursos anteriores y aplicar estos en conjunto a los de Procesamiento Avanzado de Imágenes, permitiendo entregar un análisis más profundo sobre sus resultados.
Cabe mencionar, que en general se obtuvieron buenos resultados, alcanzando un accuracy aceptable para los experimentos, variando según la cantidad de clasificadores escogidos. No obstante, estos resultados se podrían mejorar aún más realizando una cantidad mayor de experimentos, variando tanto el parámetro T como random_state, buscando el valor óptimo para estos.
Finalmente, se debe destacar la importancia de estas herramientas, las cuales permiten realizar la clasificación de personas dentro de un conjunto de imágenes. En este caso, la clasificación se realizó de manera binaria, ie, entre una clase "persona" y una "no persona", no obstante, sería interesante ampliar el análisis y comprobar el desempeño del algoritmo en una clasificación multiclase.

Referencias

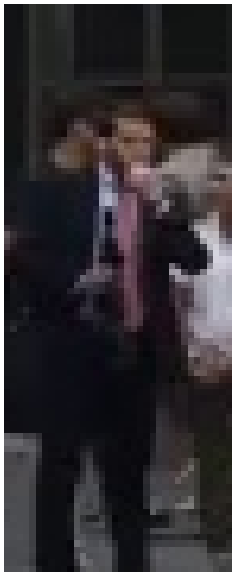
[1] First Principles of Computer Vision. Funciones de Haar para la detección de rostros, detección de rostro. 2021. Disponible en: https://www.youtube.com/watch?v=ZSqg-fZJ9tQ&t=323s

[2] First Principles of Computer Vision. Imagen Integral, detección de rostro. 2021. Disponible en: https://www.youtube.com/watch?v=5ceT8O3k6os

[3] Adakane, D. What are Haar Features used in Face Detection?. 2019. Disponible en: https://medium.com/analytics-vidhya/what-is-haar-features-used-in-face-detection-a7e531c8332b

6. Anexos

6.1. Imágenes de Ejemplo



(a) Persona

(b) Silla

(c) Auto

Figura 11: Imágenes de ejemplo

6.2. Cálculo de Haar Features

A continuación, se presenta mediante imágenes el cálculo de características Haar realizado por el autor.

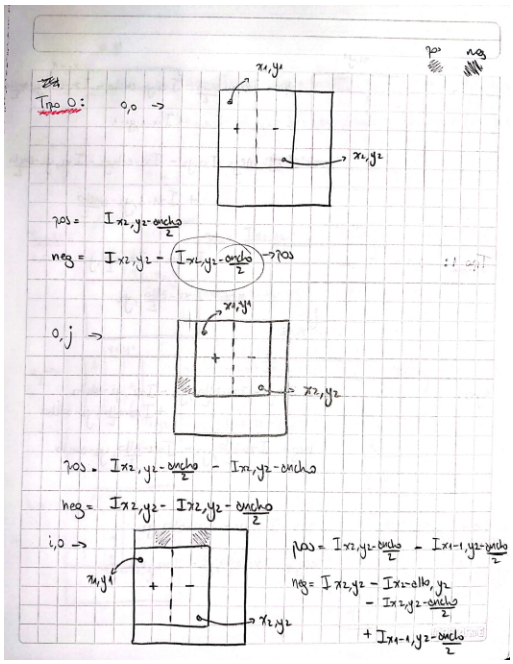


Figura 12: Cálculo de características Haar parte 1

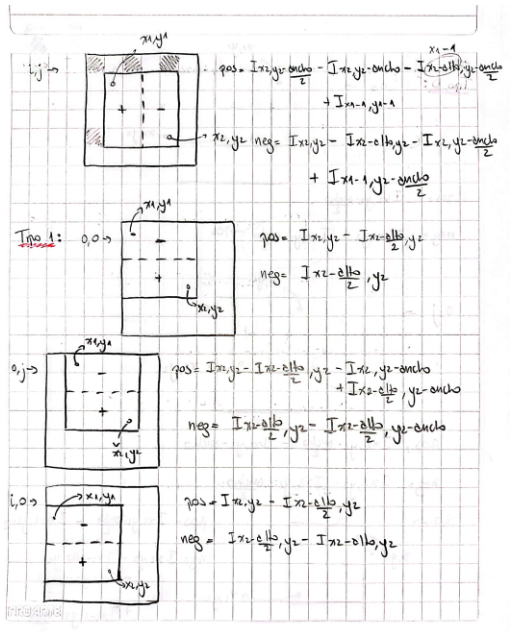


Figura 13: Cálculo de características Haar parte 2

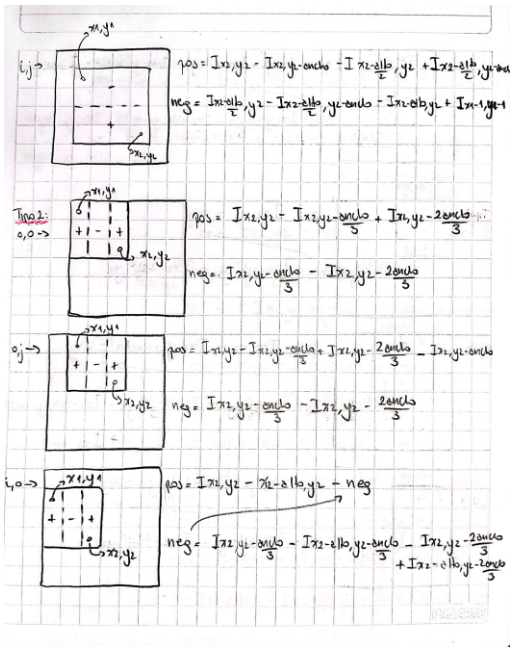


Figura 14: Cálculo de características Haar parte 3

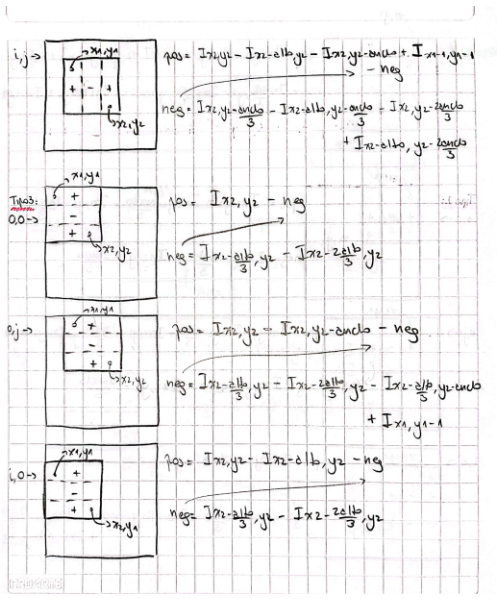


Figura 15: Cálculo de características Haar parte 4

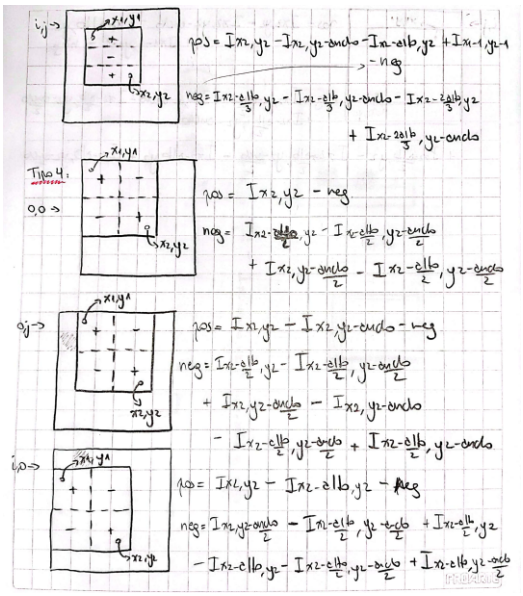


Figura 16: Cálculo de características Haar parte 5

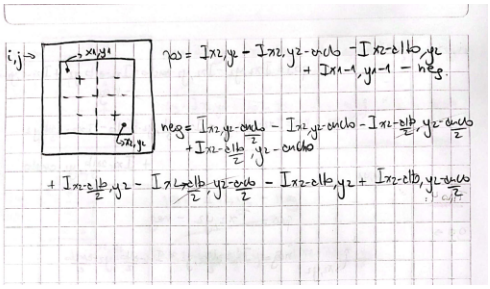


Figura 17: Cálculo de características Haar parte 6

6.3. Código Implementado

6.3.1. Parte 1

```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
import matplotlib.pyplot as plt
import glob
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import PredefinedSplit
from sklearn.model_selection import GridSearchCV
import torch
import random

!pip install ipython-autotime

%load_ext autotime
%load_ext Cython

# Nos posicionamos en la carpeta de imágenes de la tarea 4 (cabe destacar que son las mismas que la tarea 3)
%cd Procesamiento_Avanzado_de_Imagenes/ImagenesT3

# Confirmamos que estén las carpetas
!ls

def redimension(img):
  gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  # Se redimensiona la imagen
  out_img = cv2.resize(gray_img, dsize=(24, 24), interpolation = cv2.INTER_AREA)
  # Se retorna del tipo np.float32
  return np.float32(out_img)

# Se leen las imágenes recorriendo los archivos dentro de las
# Se codifica la clase como 1 para personas y como -1 para no personas

pedestrian_path = glob.glob('pedestrian//*.png') # nombres en file[11:-4]
chair_path = glob.glob('chair//*.jpg') # nombres en file[6:-4]
car_path = glob.glob('car_side//*.jpg') # nombres en file[9:-4]

Images = [] # Lista que guarda las 370 imágenes, redimensionadas a 24x24
Labels = [] # Lista que guarda las etiquetas de las imágenes

for file in pedestrian_path:
```

```
    img = cv2.imread(file)
    Images.append(redimension(img)) # se agregan las imagenes redimensionadas
    Labels.append(1) # se codifica la clase

for file in chair_path:
    img = cv2.imread(file)
    Images.append(redimension(img)) # se agregan las imagenes redimensionadas
    Labels.append(-1) # se codifica la clase

for file in car_path:
    img = cv2.imread(file)
    Images.append(redimension(img)) # se agregan las imagenes redimensionadas
    Labels.append(-1) # se codifica la clase


# Separamos los conjuntos de entrenamiento y test
X_train, X_test, Y_train, Y_test = train_test_split(Images, Labels, test_size=0.20, stratify=
    ↪ Labels, random_state = 20)
```

6.3.2. Parte 2

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

cpdef np.ndarray[np.float32_t, ndim=2] Integral_Image(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) # definimos la salida (24x24)

  cdef int i, j # Enteros para recorrer la imagen de entrada

  # Primero, analizamos los casos borde:
  # Primera posición
  output[0][0] = input[0][0]
  # 1ra Fila:
  for j in range(1, input.shape[1]): # Recorremos Columnas
    output[0][j] = output[0][j-1] + input[0][j]

  # 1ra Columna:
  for i in range(1, input.shape[0]): # Recorremos Filas
    output[i][0] = output[i-1][0] + input[i][0]

  # Ahora analizamos el resto de los casos, partiendo de la posición [1,1]
  for j in range(1, input.shape[1]):
    for i in range(1, input.shape[0]):
      output[i][j] = output[i][j-1] + output[i-1][j] - output[i-1][j-1] + input[i][j]
  return output
```

6.3.3. Parte 3

```
def Haar_Mask():
  # Para cada tipo de máscara existirán 9 diferentes combinaciones de anchos/altos
  # para esto, se comenzará en orden de tipos, ie, 0, 1, 2, 3, 4
  parameters = []
  # A continuación se mencionan las dimensiones de las máscaras (cabe destacar que las máscaras 0, 1 y 4 son de igual tamaño y las 2 y 3 entre sí igual)
  # Para las máscaras tipo 0, 1 y 4, se tendrán las siguientes combinaciones de ancho/alto:
  # 1.  4x4        5.  8x8        9.  12x12
  # 2.  4x8        6.  8x12
  # 3.  4x12       7.  12x4
  # 4.  8x4        8.  12x8

  # Para la máscara tipo 2 y 3, se tendrán las siguientes combinaciones de ancho/alto:
  # 1.  3x3        5.  6x6        9.  9x9
  # 2.  3x6        6.  6x9
  # 3.  3x9        7.  9x3
  # 4.  6x3        8.  9x6

  for i in range(0, 24, 3): # se itera 0, 3, 6, 9, 12, ..., 24 (ventana deslizante) para i y j.
    for j in range(0, 24, 3):

      x1, y1 = i, j # la esquina superior izquierda de la máscara calza con la posición i,j

      # se realizarán las 9 combinaciones para ambas polaridades usando todos los tipos, para tener todas las características posibles.
      for polaridad in [1, -1]:

        for tipo in [0, 1, 4]: # iteramos sobre los tipos que ocupen los mismos anchos/altos
          for ancho in [4, 8, 12]:
            for alto in [4, 8, 12]:
              # se define la esquina inferior derecha de la máscara considerando casos borde
              if (x1 == 0) and (y1 == 0):
                x2, y2 = x1+ancho-1, y1+alto-1
              elif (x1 == 0):
                x2, y2 = x1+ancho-1, y1+alto
              elif (y1 == 0):
                x2, y2 = x1+ancho, y1+alto-1
              else:
                x2, y2 = x1+ancho, y1+alto

              if (x2 <= 23) and (y2 <= 23): #solo si está dentro de los límites (24x24) se utiliza
                parameters.append([y1, x1, y2, x2, tipo, polaridad])

        for tipo in [2, 3]: # iteramos sobre los tipos que ocupen los mismos anchos/altos
          for ancho in [3, 6, 9]:
            for alto in [3, 6, 9]:
              # se define la esquina inferior derecha de la máscara considerando casos borde
              if (x1 == 0) and (y1 == 0):
                x2, y2 = x1+ancho-1, y1+alto-1
```

```
48            elif (x1 == 0):
49              x2, y2 = x1+ancho-1, y1+alto
50            elif (y1 == 0):
51              x2, y2 = x1+ancho, y1+alto-1
52            else:
53              x2, y2 = x1+ancho, y1+alto
54
55            if (x2 <= 23) and (y2 <= 23): #solo si está dentro de los límites (24x24) se utiliza
56              parameters.append([y1, x1, y2, x2, tipo, polaridad])
57
58  return parameters
```

6.3.4. Parte 4

```
1  def Haar_Features(images, parameters):
2    Features = [] # este arreglo guardará las características de todas las imágenes
3    for i in range(len(images)):
4      feat = [] # esta lista temporal guardará las características de c/imagen para luego guardarla en el
         ↪ arreglo principal
5      img = images[i] # imagen a la cual se aplicarán todas las máscaras parametrizadas. (debe ser
         ↪ imagen integral)
6
7      for j in range(len(parameters)):
8        # Para cada máscara parametrizada rescatamos su posición, tipo y polaridad
9        p = parameters[j]
10       y1 = p[0]
11       x1 = p[1]
12       y2 = p[2]
13       x2 = p[3]
14       tipo = p[4]
15       polaridad = p[5]
16       # A partir de sus posiciones calculamos el ancho y alto de la máscara, que nos servirá para
17       # realizar las sumas y restas adecuadas.
18       if x1 == 0:
19         alto = x2-x1+1
20       else:
21         alto = x2-x1
22       if y1 == 0:
23         ancho = y2-y1+1
24       else:
25         ancho = y2-y1
26       # Se analiza el tipo de máscara para saber que operación debe realizar el algoritmo.
27       # Para cada tipo de máscara se deben analizar los casos bordes, cuando estamos en 0,0 , en 0,j (
         ↪ con j entre 1 y 23)
28       # en i,0 (con i entre 1 y 23) y finalmente con i,j (cuando ambos son distintos de 0).
29       if tipo == 0:
30         if (x1 == 0) and (y1 == 0):
31           fpos = img[x2][y2-ancho//2]
32           fneg = img[x2][y2]-fpos
33         elif (x1 == 0):
```

```
34          fpos = img[x2][y2-ancho//2] - img[x2][y2-ancho]
35          fneg = img[x2][y2] - img[x2][y2-ancho//2]
36        elif (y1 == 0):
37          fpos = img[x2][y2-ancho//2] - img[x1-1][y2-ancho//2]
38          fneg = img[x2][y2] - img[x2-alto][y2] - fpos
39        else:
40          fpos = img[x2][y2-ancho//2] - img[x2][y2-ancho] - img[x1-1][y2-ancho//2] + img[x1-1][y1-1]
41          fneg = img[x2][y2] - img[x2-alto][y2] - img[x2][y2-ancho//2] + img[x1-1][y2-ancho//2]
42        f = (fpos - fneg) * polaridad
43
44      elif tipo == 1:
45        if (x1 == 0) and (y1 == 0):
46          fpos = img[x2][y2] - img[x2-alto//2][y2]
47          fneg = img[x2-alto//2][y2]
48        elif (x1 == 0):
49          fpos = img[x2][y2] - img[x2-alto//2][y2] - img[x2][y2-ancho] + img[x2-alto//2][y2-ancho]
50          fneg = img[x2-alto//2][y2] - img[x2-alto//2][y2-ancho]
51        elif (y1 == 0):
52          fpos = img[x2][y2] - img[x2-alto//2][y2]
53          fneg = img[x2-alto//2][y2] - img[x2-alto][y2]
54        else:
55          fpos = img[x2][y2] - img[x2][y2-ancho] - img[x2-alto//2][y2] + img[x2-alto//2][y2-ancho]
56          fneg = img[x2-alto//2][y2] - img[x2-alto//2][y2-ancho] - img[x2-alto][y2] + img[x1-1][y1-1]
57        f = (fpos - fneg) * polaridad
58
59      elif tipo == 2:
60        if (x1 == 0) and (y1 == 0):
61          fpos = img[x2][y2] - img[x2][y2-ancho//3] + img[x2][y2-2*ancho//3]
62          fneg = img[x2][y2-ancho//3] - img[x2][y2-2*ancho//3]
63        elif (x1 == 0):
64          fpos = img[x2][y2] - img[x2][y2-ancho//3] - img[x2][y2-ancho] + img[x2][y2-2*ancho//3]
65          fneg = img[x2][y2-ancho//3] - img[x2][y2-2*ancho//3]
66        elif (y1 == 0):
67          fneg = img[x2][y2-ancho//3] - img[x2-alto][y2-ancho//3] - img[x2][y2-2*ancho//3] + img[x2-
    ↪ alto][y2-2*ancho//3]
68          fpos = img[x2][y2] - img[x2-alto][y2] - fneg
69        else:
70          fneg = img[x2][y2-ancho//3] - img[x2-alto][y2-ancho//3] - img[x2][y2-2*ancho//3] + img[x2-
    ↪ alto][y2-2*ancho//3]
71          fpos = img[x2][y2] - img[x2-alto][y2] - img[x2][y2-ancho] + img[x1-1][y1-1] - fneg
72        f = (fpos - fneg) * polaridad
73
74      elif tipo == 3:
75        if (x1 == 0) and (y1 == 0):
76          fneg = img[x2-alto//3][y2] - img[x2-2*alto//3][y2]
77          fpos = img[x2][y2] - fneg
78        elif (x1 == 0):
79          fneg = img[x2-alto//3][y2] - img[x2-2*alto//3][y2] - img[x2-alto//3][y2-ancho] + img[x1][y1-1]
80          fpos = img[x2][y2] - img[x2][y2-ancho] - fneg
81        elif (y1 == 0):
82          fneg = img[x2-alto//3][y2] - img[x2-2*alto//3][y2]
```

```
83          fpos = img[x2][y2] - img[x2-alto][y2] - fneg
84        else:
85          fneg = img[x2-alto//3][y2] - img[x2-alto//3][y2-ancho] - img[x2-2*alto//3][y2] + img[x2-2*
      ↪ alto//3][y2-ancho]
86          fpos = img[x2][y2] - img[x2][y2-ancho] - img[x2-alto][y2] + img[x1-1][y1-1] - fneg
87        f = (fpos - fneg) * polaridad
88
89      elif tipo == 4:
90        if (x1 == 0) and (y1 == 0):
91          fneg = img[x2-alto//2][y2] - 2*img[x2-alto//2][y2-ancho//2] + img[x2][y2-ancho//2]
92          fpos = img[x2][y2] - fneg
93        elif (x1 == 0):
94          fneg = img[x2-alto//2][y2] - 2*img[x2-alto//2][y2-ancho//2] + img[x2][y2-ancho//2] - img[x2
      ↪ ][y2-ancho] + img[x2-alto//2][y2-ancho]
95          fpos = img[x2][y2] - img[x2][y2-ancho] - fneg
96        elif (y1 == 0):
97          fneg = img[x2][y2-ancho//2] - 2*img[x2-alto//2][y2-ancho//2] + img[x2-alto//2][y2] - img[x2-
      ↪ alto][y2] + img[x2-alto][y2-ancho//2]
98          fpos = img[x2][y2] - img[x2-alto][y2] - fneg
99        else:
100         fneg = img[x2][y2-ancho//2] - img[x2][y2-ancho] - 2*img[x2-alto//2][y2-ancho//2] + img[x2-
      ↪ alto//2][y2-ancho] + img[x2-alto//2][y2] - img[x2-alto][y2] + img[x2-alto][y2-ancho//2]
101         fpos = img[x2][y2] - img[x2][y2-ancho] - img[x2-alto][y2] + img[x1-1][y1-1] - fneg
102       f = (fpos - fneg) * polaridad
103     # Agregamos la característica a la lista variable
104     feat.append(f)
105   # cuando ya recorremos todos los parámetros para una imagen guardamos el vector y seguimos
      ↪ con la siguiente
106   Features.append(feat)
107
108 return np.array(Features)
```

6.3.5. Parte 5

```
1  def h(x,u):
2    # La función h(x,u) busca generar un clasificador débil mediante un umbral definido (u)
3    # Sea x un arreglo:
4    # x > u y x < u, devuelven un arreglo con True en las posiciones que cumplen la condición
5    lpos = x > u
6    lneg = x <= u
7    # Luego, al multiplicar estos arreglos por 1 o -1, los valores True toman este factor y los False
       ↪ quedan en 0
8    pos = lpos * 1
9    neg = lneg * -1
10   # Finalmente, si sumamos ambos arreglos (posición a posición)
11   # obtenemos un arreglo con -1 cuando x_i < u y 1 cuando x_i > u.
12   return pos + neg
13
14 def Best_U(X, y, w):
15   # X es la matriz de características, en el caso del conjunto de train, será de 296 x 3030
```

```
16  # y es el vector de clases, tendrá valores 1 o -1 según la imagen contenga una persona u otro. (largo
      ↪ 296)
17  # w es el vector de pesos de largo 296
18  # h(x,u) se construye con la función anterior
19  best_u = [] # esta lista guardará los mejores u para cada iteración.
20  best_r = [] # esta lista guardará el mejor r para cada iteración.
21  for j in range(X.shape[1]): # Recorremos las columnas de la matriz de características
22    Xcol = X[:,j] # Tomamos cada columna (ie, cada característica)
23    # Tomamos el valor mínimo y máximo
24    minX = min(Xcol)
25    maxX = max(Xcol)
26    # En cada iteración generamos una lista de r, tendrá 10 valores y luego sacaremos el mejor de
        ↪ estos.
27    r = []
28    # tendremos 10 u para cada iteración, luego guardaremos el mejor en best_u
29    u = np.linspace(minX, maxX, 10)
30
31    # Recorremos los 10 valores de u y guardamos los r correspondientes
32    for k in range(len(u)):
33      r.append(np.sum(w * y * h(Xcol, u[k])))
34    # buscamos el r máximo para cada característica
35    maxR = max(r)
36    ind = r.index(maxR)
37    # guardamos el mejor u y r en la lista correspondiente
38    best_u.append(u[ind])
39    best_r.append(maxR)
40
41  return best_u, best_r
42
43 def a_t(r):
44  return 0.5 * np.log((1+r)/(1-r))
45
46 def Train_Adaboost(X, y, T = 10):
47  N = X.shape[0] # tomamos el valor de N
48  # inicializamos el vector de pesos
49  w = np.ones(N) * (1/N)
50  # listas que guardan los elementos que permiten construir y evaluar el clasificador fuerte
51  at = np.zeros(T)
52  it = []
53  ut = np.zeros(T)
54  ht = []
55
56  # iteración principal
57  for t in range(T):
58    U, R = Best_U(X, y, w)
59
60    alphas = []
61    for i in range(len(R)):
62      alphas.append(a_t(R[i]))
63    rmax = max(R)
64    indmax = R.index(rmax) # este será el índice del elemento a guardar
```

```
65
66    # guardamos los elementos necesarios
67    at[t] = alphas[indmax]
68    it.append(indmax)
69    ut[t] = U[indmax]
70    ht.append(h(X[:,it[t]], ut[t]))
71
72    # Actualizamos los pesos
73    w = w * np.exp(-1 * at[t] * y * ht[t])
74    w = w / np.sum(w) # normalizamos los pesos
75    # se realiza la siguiente iteración...
76
77  return at, np.array(it), ut
78
79 def Adaboost_classify(X, it, ut, at):
80   # Se realiza la clasificación utilizando la función del enunciado
81   H = np.sign(np.sum(at * h(X[it], ut)))
82   return H
83
84 # Classify_all permite ingresar una matriz de características y clasificar todo directamente
85 def Classify_all(X, it, ut, at):
86   pred = []
87   for i in range(X.shape[0]):
88     pred.append(Adaboost_classify(X[i], it, ut, at))
89   return np.array(pred)
```

6.3.6. Parte 6

```
1 # Con las siguientes iteraciones se calcula la imagen integral de la imagen redimensionada
2 for img in X_train:
3   img = Integral_Image(img)
4 for img in X_test:
5   img = Integral_Image(img)
6
7 # Creamos arreglo de máscaras parametrizadas
8 parameters = Haar_Mask()
9
10 # Extraemos características
11 X_train_Features = Haar_Features(X_train, parameters)
12 X_test_Features = Haar_Features(X_test, parameters)
13
14 # Se realiza un arreglo de numpy para no tener errores
15 Y_train = np.array(Y_train)
16 Y_test = np.array(Y_test)
```

6.3.7. Parte 7

```
1 # Entrenamiento:
```

```
at, it, ut = Train_Adaboost(X_train_Features, Y_train)

# Se aplica sobre el conjunto de entrenamiento y de prueba
pred_train = Classify_all(X_train_Features, it, ut, at)
pred_test = Classify_all(X_test_Features, it, ut, at)

sn.set()
f,ax=plt.subplots()
cm = confusion_matrix(Y_train, pred_train, normalize = 'true')
sn.heatmap(cm,annot=True,ax=ax, cmap = 'Blues',xticklabels = ['persona', 'no persona'], yticklabels = ['persona', 'no persona'])

ax.set_title('Matriz de confusión Conjunto Train')
ax.set_xlabel('Predicted Labels')
ax.set_ylabel('True Labels')

print('Accuracy Conjunto Train: ', accuracy_score(Y_train, pred_train))

sn.set()
f,ax=plt.subplots()
cm = confusion_matrix(Y_test, pred_test, normalize = 'true')
sn.heatmap(cm,annot=True,ax=ax, cmap = 'Blues',xticklabels = ['persona', 'no persona'], yticklabels = ['persona', 'no persona'])

ax.set_title('Matriz de confusión Conjunto Test')
ax.set_xlabel('Predicted Labels')
ax.set_ylabel('True Labels')

print('Accuracy Conjunto Test: ', accuracy_score(Y_test, pred_test))

# Para T = 5 y T = 20 es análogo el código, no se copia al informe por esta razón
```

6.3.8. Parte 10

```
def redimension2(img):
  # Se redimensiona la imagen
  out_img = cv2.resize(gray_img, dsize=(192, 192), interpolation = cv2.INTER_AREA)
  # Se retorna del tipo np.float32
  return np.float32(out_img)

def DrawMask(img, mask_p):
  # los indices se amplifican por 8 ya que 192/24 = 8, ie, ampliamos la imagen 8 veces y por ende las máscaras igual para ser dibujadas
  y1 = mask_p[0] * 8
  x1 = mask_p[1] * 8
  y2 = mask_p[2] * 8
  x2 = mask_p[3] * 8
  tipo = mask_p[4]
  polaridad = mask_p[5]
  # A partir de sus posiciones calculamos el ancho y alto de la máscara, que nos servirá para
```

```
16  # realizar las sumas y restas adecuadas.
17  if x1 == 0:
18    alto = x2-x1+1
19  else:
20    alto = x2-x1
21  if y1 == 0:
22    ancho = y2-y1+1
23  else:
24    ancho = y2-y1
25  # colores, rojo para el lado positivo, azul para el negativo de la máscara
26  cp = (0,0,255)
27  cn = (255,0,0)
28  t = 1 # espesor
29  if tipo == 0:
30    if polaridad == 1:
31      img = cv2.rectangle(img, (x1,y1), (x2, y2-ancho//2), cp, t)
32      img = cv2.rectangle(img, (x1,y1+ancho//2), (x2, y2), cn, t)
33    else:
34      img = cv2.rectangle(img, (x1,y1), (x2, y2-ancho//2), cn, t)
35      img = cv2.rectangle(img, (x1,y1+ancho//2), (x2, y2), cp, t)
36
37  elif tipo == 1:
38    if polaridad == 1:
39      img = cv2.rectangle(img, (x1,y1), (x2-alto//2,y2), cn, t)
40      img = cv2.rectangle(img, (x1+alto//2, y1), (x2,y2), cp, t)
41    else:
42      img = cv2.rectangle(img, (x1,y1), (x2-alto//2,y2), cp, t)
43      img = cv2.rectangle(img, (x1+alto//2, y1), (x2,y2), cn, t)
44
45  elif tipo == 2:
46    if polaridad == 1:
47      img = cv2.rectangle(img, (x1,y1), (x2,y2-2*ancho//3), cp, t)
48      img = cv2.rectangle(img, (x1,y1+ancho//3), (x2,y2-ancho//3), cn, t)
49      img = cv2.rectangle(img, (x1,y1+2*ancho//3), (x2,y2), cp, t)
50    else:
51      img = cv2.rectangle(img, (x1,y1), (x2,y2-2*ancho//3), cn, t)
52      img = cv2.rectangle(img, (x1,y1+ancho//3), (x2,y2-ancho//3), cp, t)
53      img = cv2.rectangle(img, (x1,y1+2*ancho//3), (x2,y2), cn, t)
54
55  elif tipo == 3:
56    if polaridad == 1:
57      img = cv2.rectangle(img, (x1,y1), (x2-2*alto//3, y2), cp, t)
58      img = cv2.rectangle(img, (x1+alto//3, y1), (x2-alto//3, y2), cn, t)
59      img = cv2.rectangle(img, (x1+2*alto//3, y1), (x2,y2), cp, t)
60    else:
61      img = cv2.rectangle(img, (x1,y1), (x2-2*alto//3, y2), cn, t)
62      img = cv2.rectangle(img, (x1+alto//3, y1), (x2-alto//3, y2), cp, t)
63      img = cv2.rectangle(img, (x1+2*alto//3, y1), (x2,y2), cn, t)
64
65  elif tipo == 4:
66    if polaridad == 1:
```

```
      img = cv2.rectangle(img, (x1,y1), (x1+alto//2-1,y1+ancho//2-1), cp, t)
      img = cv2.rectangle(img, (x1+alto//2, y1), (x1+alto-1, y1+ancho//2-1), cn, t)
      img = cv2.rectangle(img, (x1, y1+ancho//2), (x2-alto//2, y2), cn, t)
      img = cv2.rectangle(img, (x1+alto//2,y1+ancho//2), (x2,y2), cp, t)
    else:
      img = cv2.rectangle(img, (x1,y1), (x1+alto//2-1,y1+ancho//2-1), cn, t)
      img = cv2.rectangle(img, (x1+alto//2, y1), (x1+alto-1, y1+ancho//2-1), cp, t)
      img = cv2.rectangle(img, (x1, y1+ancho//2), (x2-alto//2, y2), cp, t)
      img = cv2.rectangle(img, (x1+alto//2,y1+ancho//2), (x2,y2), cn, t)

  return img

at, it, ut = Train_Adaboost(X_train_Features, Y_train)

iMasks = it[0:5] # guardamos los primeros 5 indices


# Se toman 3 imágenes
img1 = cv2.imread('pedestrian/1.png')
img2 = cv2.imread('pedestrian/10.png')
img3 = cv2.imread('pedestrian/12.png')

img1 = redimension2(img1)
img2 = redimension2(img2)
img3 = redimension2(img3)

for index in iMasks:
  DrawMask(img1, parameters[index])

for index in iMasks:
  DrawMask(img2, parameters[index])

for index in iMasks:
  DrawMask(img3, parameters[index])

cv2_imshow(img1)
cv2_imshow(img2)
cv2_imshow(img3)
```