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

Tarea 3

Detección de Personas usando características tipo HOG

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: 07 de octubre de 2022
Fecha de entrega: 07 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 algoritmos de clasificación. Para esto, se tienen dos conceptos importantes, en primer lugar, la extracción de características, la cual puede ser en base a distintos métodos tales como usar redes convolucionales, LBP, HOG, entre otras. Luego, de extraer las características, se toma el conjunto y se subdivide en *train*, *test* y *validation* para comenzar con el segundo concepto importante, el entrenamiento y posterior clasificación del algoritmo. Este algoritmo, también puede tener diferentes orígenes, desde utilizar un Multilayer Perceptron (MLP) hasta el uso de técnicas tales como Random Forests (RF), *Support Vector Machines* (SVM) u otros tipos tanto supervisados como no supervisados. Dentro de estos conceptos mencionados anteriormente, se trabajará con la extracción de características usando HOG y una posterior clasificación usando SVM y Random Forests.

Con lo mencionado anteriormente, el presente informe tiene como objetivo el desarrollo de la tercera tarea del curso Procesamiento Avanzado de Imágenes y busca, utilizando herramientas computacionales y programación en lenguaje Python, la extracción de características mediante *Histograms of Oriented Gradients* (HOG) y la posterior clasificación usando SVM y RF, con el fin de poder detectar personas utilizando 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

Para los siguientes ítems, se debe apoyar la lectura con los códigos de la sección 6.2, ya que se hace referencia a las funciones definidas en este.

2.1. Extracción de Características HOG

Antes de extraer características de una imagen usando el método de HOG, se deben realizar los siguientes pasos.

- Realizar una redimensión de la imagen: esto es realizado utilizando la función **gray_redimension** que toma una imagen de entrada de un tamaño arbitrario, la transforma a escala de grises y la dimensiona a 128x64 píxeles.

- Cálculo de gradientes en eje x e y: esto es realizado gracias a las funciones **gradx** y **grady** extraídas de la tarea anterior, las cuales toman la imagen de entrada en escala de grises y calculan la aproximación de gradientes en base a una convolución con la máscara $[-1, 0, 1]$ y $[-1, 0, 1]^T$ respectivamente.

- Obtención de magnitud y dirección: utilizando las funciones **magnitude** y **angle**, las cuales toman los gradientes en x e y y calculan la magnitud y dirección de la siguiente manera (píxel a píxel).

$$M = \sqrt{g_x^2 + g_y^2} \tag{1}$$

$$D = \arctan\left(\frac{g_x}{g_y}\right) \tag{2}$$

- Cálculo de características: el cálculo de carácterísticas HOG, se realizó con dos funciones distintas, en primer lugar, la función **HOG_Features**, recibe la matriz de magnitud y dirección y realiza lo siguiente.
  En primer lugar, recorre la matriz de direcciones completa, analizando la posición de cada bin ($[0, 1, ..., 8)$) y cual es la ponderación que se utilizará para la magnitud. Con esto, entregará este voto pero únicamente a la celda correspondiente según recorrido, por lo tanto, esta función **no utiliza interpolación bilinear**.

  Luego, la función **HOG_Features_Bilinearly**, **si utiliza interpolación bilinear**, para esto, recorre la matriz de direcciones por parte. En primer lugar, se recorren los bloques extremos, aplicando la misma idea de la función anterior. Luego, para todos los demás bloques centrales, se utiliza un voto ponderado dividido en 8 puntos (celda, bin). Para esto, se calcula la distancia del respectivo gradiente a los centros de las celdas más cercanas (c1, c2, c3 y c4 en el código).

- Normalización: luego de obtener el arreglo de 16x8x9, se recorre este analizando bloques de 2x2 celdas normalizando estos y obteniendo un vector final de largo 2x2x15x7x9 = 3780.

- Extracción: se realiza gracias a **HOG** y **HOG_Bilinearly** que reunen todas las funciones antes mencionadas. Esta se aplica a todas las imágenes de la base de datos y se guardan tanto sus clases (0 para personas, 1 para sillas y 2 para autos), como sus vectores de características en arreglos definidos como Class y Features (Class_Bi y Features_Bi para bilinear) respectivamente.

3. Resultados

A continuación se presentan los resultados obtenidos para ambas clasificaciones, en estas, se pueden observar las matrices de confusión respectivas las cuales tienen la clase codificada como sigue.

- pedestrian: 0
- chair: 1
- car_side: 2

3.1. Clasificaciones

Para las clasificaciones se realizó la separación de conjuntos tal y como lo explicita el enunciado, en primer lugar, se separa un 20 % para test y un 80 % para train y validación (juntos). Luego, se realiza una segunda separación para los conjuntos antes mencionados, para finalizar con un 60 % para train y 20 % para validación (del total).
Luego de esto, se aplica la función PredefinedSplit para utilizar el conjunto de validación en el entrenamiento con la función GridSearchCV para encontrar la mejor combinación de hiperparámetros.

3.1.1. Clasificación Support Vector Machines

3.1.1.1. Sin Interpolación

Al aplicar GridSearch, se obtiene que el mejor modelo para el conjunto de datos extraído es SVC(kernel='sigmoid'), el cual logra alcanzar un 98.6486 % en accuracy. Su matriz de confusión se presenta a continuación.

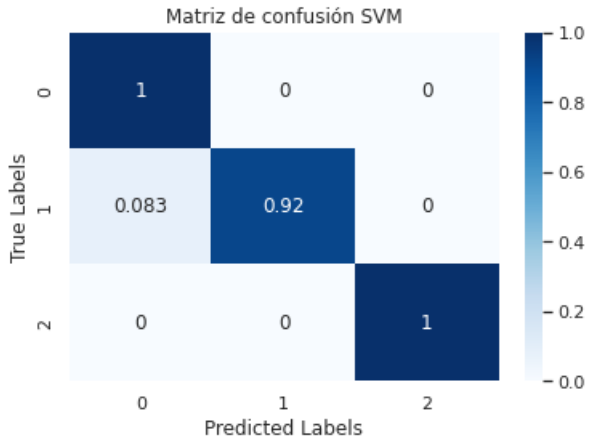


Figura 1: Matriz de confusión obtenida para la clasificación con Support Vector Machines sin interpolación

3.1.1.2. Con Interpolación

Usando GridSearch, esta vez se obtiene el modelo SVC(C=0.01, gamma=0.01, kernel='linear'), que logra alcanzar un accuracy del 94.5945 %. La matriz de confusión es la siguiente.

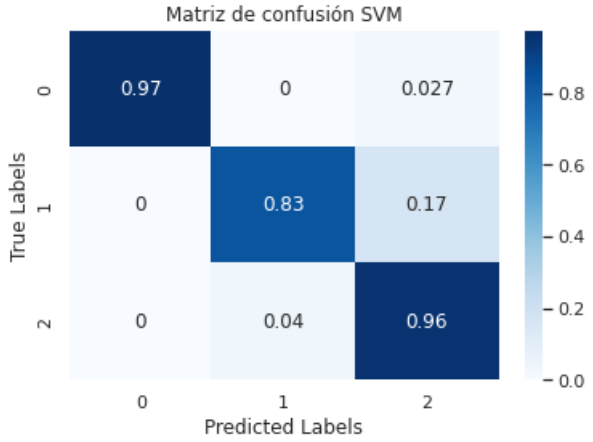


Figura 2: Matriz de confusión obtenida para la clasificación con Support Vector Machines con interpolación

3.1.2. Clasificación Random Forests

3.1.2.1. Sin Interpolación

Luego de aplicar GridSearch, se obtiene que el mejor modelo para este conjunto es RandomForestClassifier(max_depth=10, max_features='sqrt', n_estimators=50, n_jobs=-1) y logra alcanzar un 94.5946 % en accuracy. Su matriz de confusión se presenta a continuación.

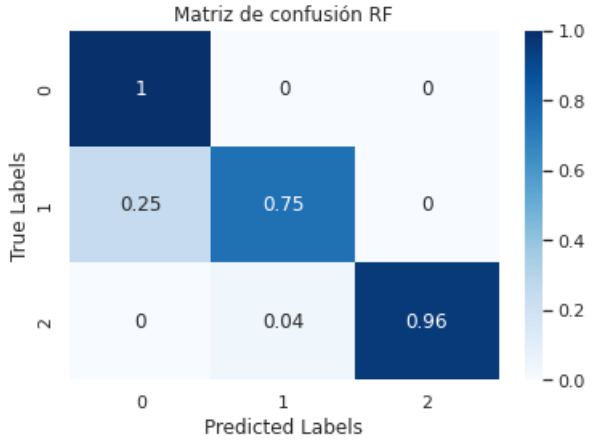


Figura 3: Matriz de confusión obtenida para la clasificación con Random Forests sin interpolación

3.1.2.2. Con Interpolación

Finalmente, para el último caso, GridSearch nos entrega el modelo RandomForestClassifier(max_depth=5, max_features='sqrt', n_estimators=50, n_jobs=-1), el cual alcanza un accuracy del 97.2973 %. La siguiente figura muestra su matriz de confusión.

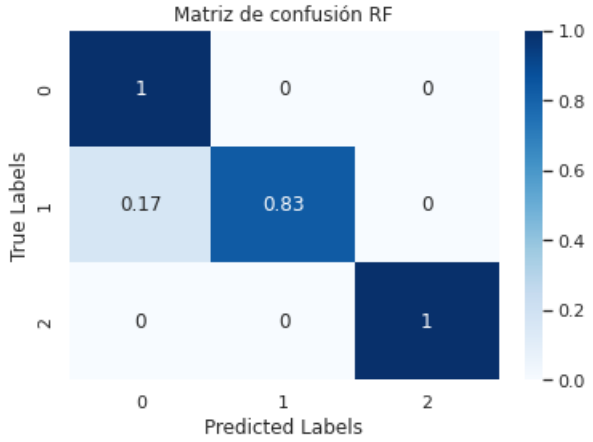


Figura 4: Matriz de confusión obtenida para la clasificación con Random Forests con interpolación

4. Análisis de Resultados

A partir de las figuras 1, 2, 3 y 4, se puede destacar los resultados positivos obtenidos con ambos métodos (con y sin interpolación bilinear), notando una clara ventaja del clasificador SVM sin interpolación, el cual alcanza una exactitud de casi el 99 %. No obstante, si volvemos al punto principal de la experiencia, basado en el reconocimiento de personas usando características HOG y clasificadores SVM y Random Forests, se puede notar que ambos tipos tanto con como sin interpolación logran obtener prácticamente el 100 % de las clasificaciones correctas para esa clase (con excepción de RF sin interpolación, no obstante, 97 % es un resultado aceptable).

A continuación se presenta un ranking de los modelos y su método HOG utilizado como forma de resumen.

1. **SVM sin interpolación:** casi 99 % de exactitud.
2. **RF con interpolación:** 97 % obtenido en accuracy test.
3. **RF sin interpolación:** 94.5946 % de exactitud.
4. **SVM con interpolación:** accuracy del 94.5945 %.

Como se puede observar, los experimentos obtienen una exactitud sobre el 90 % de los casos lo cual es bastante positivo, denotando un buen entrenamiento, lo cual se apoya firmemente en la extracción de características realizada, la cual logra diferenciar correctamente las clases estudiadas.

No obstante lo anterior, en base a las matrices de confusión se logran apreciar errores de clasificación, donde los más grandes corresponden a la imagen 2, que obtuvo la menor exactitud de los experimentos. En este, se puede notar que se detectó una cantidad considerable de personas en lugar de sillas (un cuarto de estas).
Finalmente, es interesante destacar que para el clasificador SVM, los resultados empeoran al utilizar la interpolación, mientras que para Random Forests, estos mejoran considerablemente.

5. Conclusiones

Observando los resultados obtenidos se puede afirmar que se logró cumplir el objetivo principal de la experiencia, enfocado en implementar la extracción de características HOG para su posterior clasificación utilizando dos modelos distintos, uno con relación a Support Vector Machines y otro a Random Forests. Junto a esto, se logró reforzar contenidos asociados a clasificadores, vistos tanto en el núcleo de inteligencia como en los laboratorios asociados, además, se pudo comprender el trasfondo de funciones que ya están implementadas en el lenguaje Python.
En general, los resultados fueron excelentes, logrando detectar casi el 100 % de personas en las clasificaciones, sin embargo, para esto surgieron complicaciones tales como la implementación de la interpolación bilinear, que si bien, no quedó optimizada completamente, no utiliza tantos recursos, demorando aproximadamente 3 minutos extraer las características de toda la base de datos. No obstante, esto podría ser más complejo y costoso para bases de datos más grandes.
Finalmente, cabe destacar que estas herramientas se pueden utilizar en gran variedad de problemas, notando que permiten una clasificación multiclase con buenos resultados para este tipo de conjuntos.

Referencias

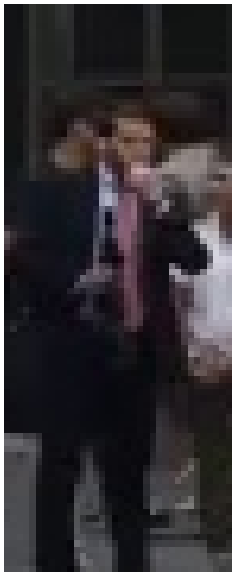
[1] Tyagi M. HOG (Histogram of Oriented Gradients): An Overview. 2021. Disponible en: https://towardsdatascience.com/hog-histogram-of-oriented-gradients-67ecd887675f

[2] Cogneethi. HOG Feature Vector Calculation. 2019. Disponible en: https://www.youtube.com/watch?v=28xk5i1_7Zc&t=188s

[3] Singh A. Feature Engineering for Images: A Valuable Introduction to the HOG Feature Descriptor. 2019. Disponible en: https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/

6. Anexos

6.1. Imágenes de Ejemplo



(a) Persona

(b) Auto

(c) Silla

Figura 5: Imágenes de ejemplo

6.2. Código Implementado

```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
import matplotlib.pyplot as plt
import glob
import pandas as pd

!pip install ipython-autotime

 %load_ext autotime

 %load_ext Cython

# Clonamos el repositorio
!git clone https://github.com/BenjaminIrarrazabal/Procesamiento_Avanzado_de_Imagenes

 %cd Procesamiento_Avanzado_de_Imagenes/ImagenesT3

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

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 int i, j # indices para recorrer la imagen de entrada

  for i in range(input.shape[0]): # se recorren todas las filas
    for j in range(1, input.shape[1] - 1): # se evitan los bordes para calzar con la aproximación [-1, 0,
      ↪ 1]
      output[i,j] = (input[i,j+1] - input[i,j-1]) # se aplica la convolución

  return output

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

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 int i, j # indices para recorrer la imagen de entrada

```

```
45   for i in range(1, input.shape[0] - 1): # se evitan los bordes para calzar con la aproximación [-1, 0,
       ↪ 1]^T
46     for j in range(input.shape[1]): # se recorren todas las columnas
47       output[i,j] = (input[i+1,j] - input[i-1,j]) # se aplica la convolución
48   return output
49
50 def gradientes(img):
51   return gradx(img), grady(img)
52
53 # gray_redimension toma una imagen de entrada y la transforma a escala de grises
54 # luego, realiza un reshape para que la imagen de salida sea del tamaño 64x128 pixeles.
55
56 def gray_redimension(img):
57   # Se transforma a escala de grises
58   gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
59   # Se redimensiona la imagen
60   out_img = cv2.resize(gray_img, dsize=(64, 128), interpolation = cv2.INTER_CUBIC)
61   return np.float32(out_img)
62
63 %%cython
64 import cython
65 import numpy as np
66 cimport numpy as np
67
68 cpdef np.ndarray[np.float32_t, ndim=2] magnitude(np.ndarray[np.float32_t, ndim=2] gx, np.ndarray
       ↪ [np.float32_t, ndim=2] gy):
69   cdef np.ndarray[np.float32_t, ndim=2] magnitude = np.zeros([gx.shape[0], gx.shape[1]], dtype = np
       ↪ .float32)
70
71   cdef int i, j # enteros para recorrer los gradientes
72
73   # Recorremos los gradientes que entran a la función
74   for i in range(magnitude.shape[0]):
75     for j in range(magnitude.shape[1]):
76       # Calculamos la magnitud
77       magnitude[i,j] = np.sqrt(gx[i,j]**2 + gy[i,j]**2)
78   return magnitude
79
80 %%cython
81 import cython
82 import numpy as np
83 cimport numpy as np
84
85 cpdef np.ndarray[np.float32_t, ndim=2] angle(np.ndarray[np.float32_t, ndim=2] gx, np.ndarray[np.
       ↪ float32_t, ndim=2] gy):
86   cdef np.ndarray[np.float32_t, ndim=2] angle = np.zeros([gx.shape[0], gx.shape[1]], dtype = np.
       ↪ float32)
87
88   cdef int i, j # enteros para recorrer los gradientes
89
90   # Recorremos los gradientes que entran a la función
```

```
  for i in range(angle.shape[0]):
    for j in range(angle.shape[1]):
      # Calculamos la dirección
      angle[i,j] = np.arctan2(gx[i,j], gy[i,j]) * 180 / np.pi
      if angle[i,j] < 0:
        angle[i,j] += 360
  return angle

def HOG_Features(magnitude, angle):
  # Se crea el arreglo de ceros
  Features = np.zeros([16, 8, 9], dtype = np.float32)
  # Se recorre el arreglo para tener todos los ángulos entre 0 y 180
  for i in range(angle.shape[0]):
    for j in range(angle.shape[1]):
      if angle[i,j] >= 180:
        angle[i,j] -= 180


  # Se recorre para extraer las características
  for i in range(angle.shape[0]):
    for j in range(angle.shape[1]):
      # k permite saber si estamos sobre un bin (0, 20, 40, 60...)
      k = angle[i,j] %20
      # m permite encontrar la posición del bin (0, 1, 2, 3....)
      m = int(angle[i,j] // 20)
      # Si k es cero, significa que toda la magnitud va al bin m
      if k == 0:
        Features[i//8, j//8, m] += magnitude[i,j]
      # si k es distinto de cero:
      else:
        # se calculan ponderaciones
        p1 = (angle[i,j] / 20) - m
        p2 = m + 1 - (angle[i,j] / 20)
        # si m es el ultimo bin, solo se agrega a este
        if m==8:
          Features[i//8, j//8, m] += (magnitude[i,j] * p2)
        # si no, se reparte según las ponderaciones obtenidas
        else:
          Features[i//8, j//8, m] += (magnitude[i,j] * p2)
          Features[i//8, j//8, m+1] += (magnitude[i,j] * p1)
  return Features

def Block_Normalization(features):
  # Se crean dos listas vacías que guardarán la información
  feature_vector = []
  normalized_vector = []
  # Se recorre el vector de 16x8x9 hasta antes de los extremos (para usar bloques de 2x2 celdas)
  for i in range(features.shape[0]-1):
    for j in range(features.shape[1]-1):
      # Se agregan las features de las celdas correspondientes
      normalized_vector.append(features[i,j])
```

```
142       normalized_vector.append(features[i,j+1])
143       normalized_vector.append(features[i+1,j])
144       normalized_vector.append(features[i+1,j+1])
145       # si el valor de las celdas es distinto a cero, se normaliza
146       if np.sum(normalized_vector) != 0:
147         normalized_vector = np.array(normalized_vector) / np.sum(normalized_vector)
148    # se retorna el vector de características de 1x3780
149       feature_vector.append(normalized_vector)
150       normalized_vector = []
151    feature_vector = np.float32(feature_vector)
152    feature_vector = feature_vector.reshape((1, 3780))
153    return np.float32(feature_vector[0])
154
155 def HOG(img):
156    # Se aplica la redimensión y transformación a escala de grises
157    img_resized = gray_redimension(img)
158    # Se computan gradientes en x e y
159    gx, gy = gradientes(img_resized)
160    # Se obtienen las matrices de magnitud y dirección (ángulos)
161    mag = magnitude(gx, gy)
162    ang = angle(gx, gy)
163    # Se extraen las características
164    FeaturesHOG = HOG_Features(mag, ang)
165    # Se normalizan features y se retorna el vector de largo 3780
166    features_vector = Block_Normalization(FeaturesHOG)
167    return features_vector
168
169 pedestrian_path = glob.glob('pedestrian//*.png') # nombres en file[11:-4]
170 chair_path = glob.glob('chair//*.jpg') # nombres en file[6:-4]
171 car_path = glob.glob('car_side//*.jpg') # nombres en file[9:-4]
172
173 Features = []
174 Class = []
175
176 for file in pedestrian_path:
177     img = cv2.imread(file)
178     feature = HOG(img)
179     Features.append(feature)
180     Class.append(0)
181
182 for file in chair_path:
183     img = cv2.imread(file)
184     feature = HOG(img)
185     Features.append(feature)
186     Class.append(1)
187
188 for file in car_path:
189     img = cv2.imread(file)
190     feature = HOG(img)
191     Features.append(feature)
192     Class.append(2)
```

```
193
194 from sklearn.model_selection import train_test_split
195 from sklearn.metrics import accuracy_score, confusion_matrix
196 from sklearn.svm import SVC
197 from sklearn.ensemble import RandomForestClassifier
198 from sklearn.preprocessing import StandardScaler
199 from sklearn.model_selection import PredefinedSplit
200
201 X_80, X_test, Y_80, Y_test = train_test_split(Features, np.array(Class), test_size=0.20, stratify=
        ↪ Class)
202
203 X_train, X_val, Y_train, Y_val = train_test_split(X_80, Y_80, train_size=0.75)
204
205 scaler = StandardScaler()
206 X_train = scaler.fit_transform(X_train)
207 X_80 = scaler.transform(X_80)
208 X_test = scaler.transform (X_test)
209
210 test_fold = [-1 for _ in range(int(len(X_80)*0.75))] + [0 for _ in range(int(len(X_80)*0.25))]
211
212 PSplit = PredefinedSplit(test_fold)
213
214 from sklearn.model_selection import GridSearchCV
215 # Grid Search para Support Vector Machines
216 model = SVC()
217 parametros = {'C': [0.01, 0.1, 1.0, 10, 100, 1000],
218               'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
219               'gamma': [0.01, 0.1, 1.0, 10, 'scale', 'auto']}
220
221 grid = GridSearchCV(model, parametros, cv=PSplit)
222 grid.fit(X_80, Y_80)
223 grid.best_estimator_
224
225 # Clasificación usando el mejor modelo encontrado
226 classifier = SVC(kernel = 'sigmoid')
227 classifier.fit(X_train, Y_train)
228
229 Y_pred = classifier.predict(X_test)
230 print('Accuracy Test SVM: ', accuracy_score(Y_test, Y_pred))
231
232 import seaborn as sn
233 sn.set()
234 f,ax=plt.subplots()
235 cm = confusion_matrix(Y_test, Y_pred, normalize = 'true')
236 sn.heatmap(cm,annot=True,ax=ax, cmap = 'Blues')
237
238 ax.set_title('Matriz de confusión SVM')
239 ax.set_xlabel('Predicted Labels')
240 ax.set_ylabel('True Labels')
241
242 from sklearn.model_selection import GridSearchCV
```

```
# Grid Search para RandomForest
model = RandomForestClassifier()
parametros = {'n_estimators': [20, 50, 70, 100, 130, 180],
              'criterion': ['gini', 'entropy', 'log_loss'],
              'max_depth': [5, 10, 20, 30, 40, 100],
              'max_features': ['sqrt', 'log2'],
              'n_jobs': [1, -1]}
grid = GridSearchCV(model, parametros, cv=PSplit)
grid.fit(X_80, Y_80)
grid.best_estimator_

classifier = RandomForestClassifier(max_depth=10, max_features='sqrt', n_estimators=50, n_jobs
    ↪ =-1)
classifier.fit(X_train, Y_train)

Y_pred = classifier.predict(X_test)
print('Accuracy Test RF: ', accuracy_score(Y_test, Y_pred))

import seaborn as sn
sn.set()
f,ax=plt.subplots()
cm = confusion_matrix(Y_test, Y_pred, normalize = 'true')
sn.heatmap(cm,annot=True,ax=ax, cmap = 'Blues')

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

def HOG_Features_Bilinearly(magnitude, angle):
  # Se crea el arreglo de features
  Features = np.zeros([16, 8, 9], dtype = np.float32)
  # Se recorren los ángulos para que estén entre 0 y 180
  for i in range(angle.shape[0]):
    for j in range(angle.shape[1]):
      if angle[i,j] >= 180:
        angle[i,j] -= 180
  # Se recorren los extremos y se aplica el mismo funcionamiendo que HOF_Features
  # es decir, no se usa la interpolación
  for i in range(angle.shape[0]-8, angle.shape[0]):
    for j in range(angle.shape[1]-8, angle.shape[1]):
      k = angle[i,j]%20
      m = int(angle[i,j] // 20)
      if k == 0:
        Features[i//8, j//8, m] += magnitude[i,j]
      else:
        p1 = (angle[i,j] / 20) - m
        p2 = m + 1 - (angle[i,j] / 20)
        if m==8:
          Features[i//8, j//8, m] += (magnitude[i,j] * p2)
        else:
          Features[i//8, j//8, m] += (magnitude[i,j] * p2)
```

```
293         Features[i//8, j//8, m+1] += (magnitude[i,j] * p1)
294  # Para los casos generales se aplica la interpolación de la siguiente manera
295  for i in range(angle.shape[0]-8):
296    for j in range(angle.shape[1]-8):
297      # k y m funcionan de igual manera que antes
298      k = angle[i,j]%20
299      m = int(angle[i,j] // 20)
300
301      p1 = (angle[i,j] / 20) - m        #  Ponderación bin más lejano
302      p2 = m + 1 - (angle[i,j] / 20)    #  Ponderación bin más cercano
303
304      # Las celdas más cercanas se determinan mediante un bloque de 2x2
305      c1 = Features[i//8, j//8]         # Celda directamente más cercana
306      c2 = Features[i//8, (j//8)+1]     # Celda de la derecha
307      c3 = Features[(i//8)+1, j//8]     # Celda de abajo
308      c4 = Features[(i//8)+1, (j//8)+1] # Celda diagonal
309
310      # Bines
311      b1 = m
312      b2 = m + 1
313
314      if (i%4 == 0) and (j%4 == 0): # Si está en el centro de la celda
315      # Si está en el centro de la celda, toda la ponderación de magnitud debe ir a esta celda en
         ↪ particular y luego realizar
316      # la ponderación interna entre bines
317        if m == 8:
318          c1[b1] += (magnitude[i,j] * p2)
319        else:
320          c1[b1] += (magnitude[i,j] * p2)
321          c1[b2] += (magnitude[i,j] * p1)
322
323      elif (i%7 == 0) and (j%7 == 0): # Si está entre 4 celdas
324      # Si está entre 4 celdas (justo en la unión), se reparte equitativamente y posteriormente se
         ↪ pondera internamente
325        if m == 8:
326          c1[b1] += (magnitude[i,j] * p2) * 0.25
327          c2[b1] += (magnitude[i,j] * p2) * 0.25
328          c3[b1] += (magnitude[i,j] * p2) * 0.25
329          c4[b1] += (magnitude[i,j] * p2) * 0.25
330        else:
331          c1[b1] += (magnitude[i,j] * p2) * 0.25
332          c2[b1] += (magnitude[i,j] * p2) * 0.25
333          c3[b1] += (magnitude[i,j] * p2) * 0.25
334          c4[b1] += (magnitude[i,j] * p2) * 0.25
335
336          c1[b2] += (magnitude[i,j] * p1) * 0.25
337          c2[b2] += (magnitude[i,j] * p1) * 0.25
338          c3[b2] += (magnitude[i,j] * p1) * 0.25
339          c4[b2] += (magnitude[i,j] * p1) * 0.25
340      else: # Cualquier otro caso
341      # En el caso general, se calculan las 4 distancias d1,..., d4
```

```
342       # estas se intercambian debidamente para ponderar las magnitudes de cada celda y bin
          ↪ respectivo
343       # El resto es análogo a todo lo anterior
344         d1 = np.abs(i-(i//8 * 8 + 4)) + np.abs(j-(j//8 * 8 + 4))
345         d2 = np.abs(i-(i//8 * 8 + 4)) + np.abs(j-(j//8 * 8 + 8))
346         d3 = np.abs(i-(i//8 * 8 + 8)) + np.abs(j-(j//8 * 8 + 4))
347         d4 = np.abs(i-(i//8 * 8 + 8)) + np.abs(j-(j//8 * 8 + 8))
348         t = d1+d2+d3+d4
349         if m == 8:
350           if d2 > d3:
351             c1[b1] += (magnitude[i,j] * p2) * (d4/t)
352             c2[b1] += (magnitude[i,j] * p2) * (d3/t)
353             c3[b1] += (magnitude[i,j] * p2) * (d2/t)
354             c4[b1] += (magnitude[i,j] * p2) * (d1/t)
355           else:
356             c1[b1] += (magnitude[i,j] * p2) * (d4/t)
357             c2[b1] += (magnitude[i,j] * p2) * (d2/t)
358             c3[b1] += (magnitude[i,j] * p2) * (d3/t)
359             c4[b1] += (magnitude[i,j] * p2) * (d1/t)
360
361         else:
362           if d2 > d3:
363             c1[b1] += (magnitude[i,j] * p2) * (d4/t)
364             c2[b1] += (magnitude[i,j] * p2) * (d3/t)
365             c3[b1] += (magnitude[i,j] * p2) * (d2/t)
366             c4[b1] += (magnitude[i,j] * p2) * (d1/t)
367
368             c1[b2] += (magnitude[i,j] * p1) * (d4/t)
369             c2[b2] += (magnitude[i,j] * p1) * (d3/t)
370             c3[b2] += (magnitude[i,j] * p1) * (d2/t)
371             c4[b2] += (magnitude[i,j] * p1) * (d1/t)
372           else:
373             c1[b1] += (magnitude[i,j] * p2) * (d4/t)
374             c2[b1] += (magnitude[i,j] * p2) * (d2/t)
375             c3[b1] += (magnitude[i,j] * p2) * (d3/t)
376             c4[b1] += (magnitude[i,j] * p2) * (d1/t)
377
378             c1[b2] += (magnitude[i,j] * p1) * (d4/t)
379             c2[b2] += (magnitude[i,j] * p1) * (d2/t)
380             c3[b2] += (magnitude[i,j] * p1) * (d3/t)
381             c4[b2] += (magnitude[i,j] * p1) * (d1/t)
382
383   return Features
384
385 def HOG_Bilinearly(img):
386   # Se aplica la redimensión y transformación a escala de grises
387   img_resized = gray_redimension(img)
388   # Se computan gradientes en x e y
389   gx, gy = gradientes(img_resized)
390   # Se obtienen las matrices de magnitud y dirección (ángulos)
391   mag = magnitude(gx, gy)
```

```
392   ang = angle(gx, gy)
393   # Se extraen las características
394   FeaturesHOG = HOG_Features_Bilinearly(mag, ang)
395   # Se normalizan features y se retorna el vector de largo 3780
396   features_vector = Block_Normalization(FeaturesHOG)
397   return features_vector
398
399 # El resto es análogo (ver notebook ipynb)
```