

Universidad ORT Uruguay

Facultad de Ingeniería

OBLIGATORIO

MACHINE LEARNING

Mateo Goldwasser - 239420

Juan Diego Etcheverry - 252443

Nicolas Edelman - 251363

Tutor: Federico Young, Federico Vilensky
2023

Introducción:	1
1. Funciones auxiliares	2
Extractor de patches	2
Sintetizador de rectángulos	2
TPR & FPR	3
2. Pre-Procesamientos	4
Inicializar el Dataset de caras	4
Inicializar el Dataset de Backgrounds	4
Matriz de Features	4
Extracción de Features	5
3. Definir modelos a utilizar	6
4. Modelo de cascada	9
5. Función principal	12
1. Inicialización	12
2. Iteración	12
3. Procesamiento del patch	13
4. Supresión	13

Introducción:

El objetivo del obligatorio es aplicar las técnicas de clasificación vistas en el curso al problema de detección de objetos en imágenes. Concretamente nos enfocaremos en la detección de rostros, buscando encontrar un modelo que nos permita predecir de forma eficiente, precisa y eficaz, todas las imágenes de una cara.

Estructura del proyecto:

El proyecto se dividió en 3 partes fundamentales:

1. Funciones auxiliares
2. Pre-procesamientos
3. Definir los modelos a utilizar
4. Crear mi modelo de cascada
5. Crear la función principal

1. Funciones auxiliares

Extractor de patches

Dado una imagen “img”, se extraen muchas porciones de esta de distintas escalas. Esto lo usamos para dar un set de imágenes de “no caras”, generar muchísimas más que tampoco lo serán. Tener muchas de estas nos sirve para tener más ejemplos a la hora de entrenar nuestros modelos.

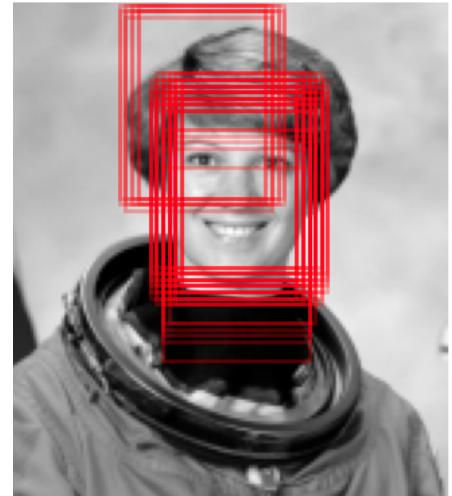
```
def extract_patches(img, N, scale=1.0, patch_size=(62,47)):
    H = img.shape[0]
    W = img.shape[1]
    H_patch = min(H , int(scale * patch_size[0]))
    W_patch = min(W , int(scale * patch_size[1]))
    extracted_patch_size = (H_patch, W_patch)
    extractor = PatchExtractor(patch_size=extracted_patch_size, max_patches=N, random_state=0)
    patches = extractor.transform(img[np.newaxis])
    if scale != 1:
        patches = np.array([resize(patch, patch_size) for patch in patches])
    return patches
```

Sintetizador de rectángulos:

Una vez que tengamos nuestra función principal, esperamos que ésta registre todos los rectángulos que el modelo afirma ser una cara. Esto lo que puede provocar es que pase algo como en la siguiente imagen.

Para sintetizar esto, hacemos uso de una función que justamente sintetiza los rectángulos en uno solo.

Lo ideal en esta función es que el tamaño y la posición final de este único rectángulo resultante sea una especie de promedio ponderado entre todos los rectángulos que se están suprimiendo.



```

def non_max_suppression1(detections, overlapThresh):
    # Si no hay detecciones, regresar una lista vacía
    if len(detections) == 0:
        return []

    # Convertir las detecciones a un arreglo de NumPy
    indices = np.array([d[0], d[1], d[0] + d[2], d[1] + d[3]] for d in detections), dtype=float)

    # Inicializar la lista de detecciones seleccionadas
    pick = []

    # Tomar las coordenadas de los cuadros
    x1 = indices[:, 0]
    y1 = indices[:, 1]
    x2 = indices[:, 2]
    y2 = indices[:, 3]

    # Calcula el área de los cuadros y ordena los cuadros
    area = (x2 - x1 + 1) * (y2 - y1 + 1)
    idxs = np.argsort(area)

    # Mientras todavía hay índices en la lista de índices
    while len(idxs) > 0:
        # Toma el último índice de la lista y agrega el índice a la lista de seleccionados
        last = len(idxs) - 1
        i = idxs[last]
        pick.append(i)

        # Encontrar las coordenadas (x, y) más grandes para el inicio de la caja y las coordenadas (x, y) más pequeñas para el final de la caja
        xx1 = np.maximum(x1[i], x1[idxs[:last]])
        yy1 = np.maximum(y1[i], y1[idxs[:last]])
        xx2 = np.minimum(x2[i], x2[idxs[:last]])
        yy2 = np.minimum(y2[i], y2[idxs[:last]])

        # Calcula el ancho y alto de la caja
        w = np.maximum(0, xx2 - xx1 + 1)
        h = np.maximum(0, yy2 - yy1 + 1)

        # Calcula la proporción de superposición
        overlap = (w * h) / area[idxs[:last]]

        # Elimina todos los índices del índice de lista que tienen una proporción de superposición mayor que el umbral proporcionado
        idxs = np.delete(idxs, np.concatenate(([last], np.where(overlap > overlapThresh)[0])))

    # Devuelve solo las detecciones seleccionadas
    return [detections[i] for i in pick]

```

TPR & FPR:

TPR describe la tasa a la que el clasificador predice como “positivo” a las observaciones que son “positivas”. FPR describe la tasa a la que el clasificador predice como “positivo” a las observaciones que en realidad son “negativas”.

```

# True Positive Rate
def tpr_scorer(clf, X, y):
    y_pred = clf.predict(X)
    cm = confusion_matrix(y, y_pred)
    tpr = cm[1,1]/(cm[1,1]+cm[1,0])
    return tpr

```

```

# False Positive Rate
def fpr_scorer(clf, X, y):
    y_pred = clf.predict(X)
    cm = confusion_matrix(y, y_pred)
    fpr = cm[0,1]/(cm[0,0]+cm[0,1])
    return fpr

```

2. Pre-Procesamientos:

Inicializar el Dataset de caras:

Esta función nos devuelve un total de 13.233 fotos de rostros que tienen un tamaño de 62px de alto y 47px de ancho. Con estas fotos es que entrenaremos el modelo.

```
# Cargamos el dataset de caras
faces = fetch_lfw_people()
positive_patches = faces.images
positive_patches.shape
✓ 0.5s
(13233, 62, 47)
```

Inicializar el Dataset de Backgrounds:

```
# Extraemos las imágenes de fondo
negative_patches = np.vstack(
    [extract_patches(im, num_patches, scale)
     for im in tqdm(backgrounds, desc='Procesando imágenes')
     for scale in scales]
)
negative_patches.shape
✓ 2m 2.7s
Procesando imágenes: 100%|██████████| 26/26 [01:51<00:00,  4.28s/it]
(124949, 62, 47)
```

Lo que hicimos fue tomar algunas imágenes de la librería sklearn y algunas imágenes caseras adicionales dadas por los profesores. En total suman 26 imágenes, pero como explicamos anteriormente, haciendo uso de la función de extraer patches, logramos generar unas 124.949 con las mismas dimensiones que las caras para usar.

Matriz de Features:

El objetivo de este punto, es poder exponer en una matriz “X” todas las imágenes recopiladas en el paso anterior (positive_patches + negative_patches) a cada una extraerle las features para poder analizarlas. A su vez, se tendrá un array aparte “y” donde irán las etiquetas de si la imagen en cuestión es una cara o no Debería quedar algo así:

$$\begin{array}{c} \mathbf{X} = \begin{matrix} & f_1 & f_2 & f_3 & \dots & f_M \\ \text{imagen 1} & | & | & | & & | \\ \text{imagen 2} & | & | & | & & | \\ \text{imagen 3} & | & | & | & & | \\ \vdots & | & | & | & & | \\ \text{imagen } N & | & | & | & & | \end{matrix} \\ \mathbf{y} = \begin{matrix} \text{etiqueta 1} \\ \text{etiqueta 2} \\ \text{etiqueta 3} \\ \vdots \\ \text{etiqueta } N \end{matrix} \end{array}$$

Extracción de Features:

Para la extracción de features, hicimos uso de lo que se denominan las HOG features (Histogram of Oriented Gradients). Este es un método que se usa para determinar si hay determinados objetos en una imagen.

Para esto, lo primero que se hace es dividir la imagen en celdas más pequeñas. Para cada píxel en cada celda, se calculan los gradientes horizontal y vertical, que no son más que una representación de la diferencia de luz (entrante y saliente) que hay en la celda.

Luego, los gradientes calculados se agrupan en celdas. Cada celda guarda información sobre la orientación de los gradientes en esa región específica. Luego de algunas normalizaciones, se agrupan varias celdas en bloques, y son estos los que se usan para formar el vector de características HOG que se usa en la matriz.

Para nuestro obligatorio, usaremos 81 features, que se ponen en el eje horizontal en la matriz "X".

Matriz de Features

```
# Armamos la matriz de features y el vector de etiquetas
X = np.array(
    [feature.hog(image=im,
                  orientations=orientations,
                  pixels_per_cell=pixels_per_cell,
                  cells_per_block=cells_per_block)
     for im in tqdm(chain(positive_patches, negative_patches))])
y = np.zeros(X.shape[0])
y[:positive_patches.shape[0]] = 1

print("Matriz de características: ", X.shape)
print("Array de etiquetas: ", y.shape)
✓ 22.4s
```

Las orientaciones, píxeles por celda y celdas por bloque son parámetros que se definen en los modelos que explicaremos a continuación.

Como dijimos, el resultado de esto es justamente la matriz de Features de la suma de positive y negative patches de alto y las 81 HOG features de ancho.

```
Matriz de características: (138182, 81)
Array de etiquetas: (138182,)
```

3. Definir modelos a utilizar:

En esta parte, la idea fue ir probando los distintos modelos vistos en clase y para cada modelo, ir cambiando los hiperparametros. Con cada hiper-parámetro, corrimos con un cross-validation, el cual nos daría distintos scores, como el tiempo de entrenamiento, el accuracy, la precisión, recall, etc.

Cada prueba la fuimos registrando en una tabla, para luego poder discutir con el equipo acerca de cuáles hiperparametros nos convienen más para cada modelo.

Para la decisión tuvimos en cuenta muchos factores. Entre ellos;

- Tiempo de entrenamiento
- Accuracy
- Precision
- Recall
- F1
- AUC
- TPR
- FPR

Para decidir, era importante hacer un trade-off entre calidad y tiempo, ya que un modelo ultra complejo con una accuracy y precisión de 0,99 no me sirve para el análisis de una imagen si demora 1 minuto en decidir si es una cara o no. Lo mismo al revés, no me sirve un modelo en 0,1 segundo de un resultado, pero con 0,50 de TPR.

A continuación presentamos las tablas de los modelos y sus hiper-parámetros utilizados.

RandomForest										
n_estimators	max_depth	Time	Accuracy	Precision	Recall	F1	B_Accuracy	AUC	TPR	FPR
10	3	2.348648	0.949	0.954	0.744507	0.810662	0.744507	0.980	0.491423	0.002408
20	3	4.758757	0.950891	0.961335	0.750014	0.817524	0.750014	0.983569	0.501549	0.001529
40	3	9.28679	0.947736	0.966232	0.729993	0.799881	0.729993	0.988546	0.460666	0.00068
80	3	18.826663	0.945152	0.967148	0.715389	0.785668	0.715389	0.989642	0.431194	0.000418
200	3	47.31202	0.942308	0.966778	0.700033	0.769756	0.700033	0.991099	0.400361	0.000296
500	3	118.194027	0.941838	0.966864	0.697442	0.767054	0.697442	0.991442	0.395147	0.000264
1000	3	235.912319	0.940904	0.966698	0.692432	0.761677	0.692432	0.991772	0.385096	0.000232

Regresion Logistica										
penalty	C	Time	Accuracy	Precision	Recall	F1	B_Accuracy	AUC	TPR	FPR
I2	1	1.872234	0.98824	0.967645	0.966876	0.966668	0.966876	0.9967	0.940451	0.006699
I2	2	4.913905	0.988309	0.967001	0.968518	0.987201	0.968518	0.996753	0.943927	0.006691
I2	3	2.582777	0.988421	0.966555	0.969273	0.967309	0.969273	0.996775	0.94559	0.007043
I2	4	2.593705	0.98845	0.966099	0.969931	0.967422	0.969931	0.99679	0.947026	0.007163
none	1	8.51697	0.988537	0.964976	0.971534	0.967702	0.971534	0.996904	0.950502	0.007435
none	2	2.420642	0.98845	0.966099	0.969931	0.967422	0.969931	0.99679	0.947026	0.007163
none	3	7.158566	0.988537	0.964976	0.971534	0.967702	0.971534	0.996904	0.950502	0.007435
none	4	7.858798	0.988537	0.964976	0.971534	0.967702	0.971534	0.996904	0.950502	0.007435

KNN										
K	Time	Accuracy	Precision	Recall	F1	B_Accuracy	AUC	TPR	FPR	
1	0.022	0.984	0.942	0.980	0.959	0.980142	0.980142	0.974987	0.014702	
2	0.021	0.989	0.965	0.976	0.969	0.976362	0.987841	0.960855	0.008131	
3	0.020	0.986	0.951	0.981	0.964	0.980558	0.990241	0.973248	0.012133	
5	0.020	0.987	0.953	0.980	0.965	0.979984	0.992254	0.971284	0.011317	
10	0.019	0.987	0.953	0.980	0.965	0.979984	0.992254	0.971284	0.011317	
20	0.020	0.987	0.957	0.976	0.965	0.975544	0.996451	0.981309	0.01022	
40	0.021	0.987	0.957	0.973	0.964	0.972818	0.997096	0.955641	0.010004	
500	0.017	0.985	0.963	0.955	0.958	0.955002	0.996668	0.917327	0.007323	
2000	0.026	0.983	0.970	0.930	0.948	0.929555	0.995145	0.863823	0.004714	

KNN										
K	Time	Accuracy	Precision	Recall	F1	B_Accuracy	AUC	TPR	FPR	
1	0.022	0.984	0.942	0.980	0.959	0.980142	0.980142	0.974987	0.014702	
2	0.021	0.989	0.965	0.976	0.969	0.976362	0.987841	0.960855	0.008131	
3	0.020	0.986	0.951	0.981	0.964	0.980558	0.990241	0.973248	0.012133	
5	0.020	0.987	0.953	0.980	0.965	0.979984	0.992254	0.971284	0.011317	
10	0.019	0.987	0.953	0.980	0.965	0.979984	0.992254	0.971284	0.011317	
20	0.020	0.987	0.957	0.976	0.965	0.975544	0.996451	0.981309	0.01022	
40	0.021	0.987	0.957	0.973	0.964	0.972818	0.997096	0.955641	0.010004	
500	0.017	0.985	0.963	0.955	0.958	0.955002	0.996668	0.917327	0.007323	
2000	0.026	0.983	0.970	0.930	0.948	0.929555	0.995145	0.863823	0.004714	

Arboles de decision										
Profundidad	Criterio	Time	Accuracy	Precision	Recall	F1	B_Accuracy	AUC	TPR	FPR
1	Entropia	1.120988	0.904235	0.452117	0.5	0.474855	0.5	0.77589	0	0
2	Entropia	2.12247	0.93277	0.806325	0.80783	0.806781	0.80783	0.889416	0.653291	0.037631
3	Entropia	3.200729	0.94822	0.882653	0.79861	0.832229	0.79861	0.944696	0.809082	0.015662
4	Entropia	4.045833	0.953069	0.869119	0.862162	0.863853	0.862162	0.964768	0.749718	0.025394
5	Entropia	4.959466	0.963693	0.900603	0.889555	0.894405	0.889555	0.974453	0.797854	0.018744

Ada Boost										
n_estimators	learning_rate	Time	Accuracy	Precision	Recall	F1	B_Accuracy	AUC	TPR	FPR
4	0.5	4.850711	0.94529	0.917468	0.74293	0.801716	0.74293	0.945884	0.492631	0.006771
6	0.5	7.310285	0.941881	0.954711	0.702499	0.770615	0.702499	0.971616	0.406406	0.001409
8	0.5	9.433653	0.943353	0.941897	0.817796	0.886007	0.817796	0.98049	0.642186	0.006596
10	0.5	11.997219	0.965075	0.946263	0.844949	0.886899	0.844949	0.98619	0.696365	0.006467

Bagging										
N° Estimadores	Time	Accuracy	Precision	Recall	F1	B_Accuracy	AUC	TPR	FPR	
1	5.226564	0.968555	0.901626	0.851342	0.815634	0.851342	0.851342	0.884677	0.021593	
2	12.427473	0.979129	0.963992	0.912448	0.936272	0.912448	0.96377	0.829971	0.005074	
3	19.037345	0.97931	0.938711	0.945318	0.941452	0.945318	0.97427	0.903272	0.012637	
4	25.195105	0.982791	0.963752	0.935857	0.949039	0.935857	0.980007	0.877805	0.00609	
5	31.733028	0.981894	0.948706	0.950023	0.948692	0.950023	0.983157	0.910602	0.010556	

Grad Boosting										
n_estimators	learning_rate	Time	Accuracy	Precision	Recall	F1	B_Accuracy	AUC	TPR	FPR
1	0.3	3.307141	0.94822	0.882653	0.79861	0.832229	0.79861	0.944996	0.609082	0.015882
2	0.3	6.396877	0.953482	0.931363	0.78462	0.838974	0.78462	0.969151	0.575755	0.006515
4	0.3	12.998861	0.962636	0.946057	0.828906	0.876091	0.828906	0.981546	0.663493	0.005682
6	0.3	19.102515	0.968013	0.947023	0.861505	0.89829	0.861505	0.987046	0.729765	0.006755
1	0.5	3.173761	0.94822	0.882653	0.79861	0.832229	0.79861	0.944996	0.609082	0.015882
2	0.5	6.335992	0.950927	0.883137	0.816011	0.84514	0.816011	0.968383	0.649134	0.017111
4	0.5	12.699111	0.965857	0.918988	0.877778	0.896695	0.877778	0.98196	0.768834	0.013277
6	0.5	19.195878	0.973846	0.941237	0.905303	0.921804	0.905303	0.988717	0.820522	0.009916
1	0.8	3.212862	0.941621	0.848762	0.815871	0.826163	0.815871	0.944996	0.65541	0.028057
2	0.8	6.386224	0.955609	0.876052	0.868127	0.871385	0.868127	0.958867	0.75992	0.023666
4	0.8	12.807184	0.965915	0.910158	0.894432	0.901142	0.894432	0.977561	0.806015	0.017151
6	0.8	19.156368	0.974975	0.934114	0.922853	0.927668	0.922853	0.986354	0.858383	0.012677

Red Neuronal										
layers	Neurons per layer	Time	Accuracy	Precision	Recall	F1	B_Accuracy	AUC	TPR	FPR
1	1	0.324	0.906	0.991	0.544	0.46	0.611	0.622	0.245	0.1
1	5	0.224	0.906	0.991	0.544	0.46	0.611	0.798	0.245	0.1
2	1	0.523	0.906	0.991	0.544	0.46	0.611	0.761	0.245	0.1
2	5	0.887	0.923	0.935	0.916	0.931	0.924	0.991	0.887	0.022
3	5	0.971	0.977	0.904	0.947	0.942	0.958	0.991	0.889	0.003
3	20	3.792	0.991	0.923	0.972	0.968	0.973	0.995	0.901	0.001
3	30	4.904	0.981	0.927	0.966	0.966	0.961	0.994	0.913	0.001
5	20	5.339	0.983	0.919	0.969	0.948	0.943	0.996	0.895	0.003

En muchas ocasiones, planeamos trabajar con hiperparametros un tanto mayores para ver qué pasaba, pero los tiempos ya eran ridículamente altos, por lo que decidimos cortar antes.

Para ejecutar estas pruebas, se fue ejecutando para cada modelo y cada hiper parámetro los siguientes scripts.

```
model = LogisticRegression(penalty='l2', C=4, max_iter=1000)
resolution = 1
scales = [0.5, 1, 2, 4, 8]
proportion = 10
num_patches = int((proportion * len(positive_patches)) / (len(scales) * len(backgrounds)))
orientations = 3
pixels_per_cell = (12, 12)
cells_per_block = (3, 3)

model_name = str(model)
experiment_name = model_name
experiment_name += '_R_' + str(resolution)
experiment_name += '_S_' + str(scales)
experiment_name += '_P_' + str(proportion)
experiment_name += '_O_' + str(orientations)
experiment_name += '_C_' + str(pixels_per_cell)
experiment_name += '_B_' + str(cells_per_block)

print(experiment_name)
```

Se definen como dijimos la resolución de la imagen, las escalas que se van a utilizar, la proporción y más datos. Una vez finalizada nuestra función principal probamos en cambiar algunos de estos parámetros, como las orientaciones y las escalas, pero llegamos a que la mejor combinación era sin lugar a dudas la de la imagen.

```
# Diccionario de scores
model.fit(X,y)
scoring = {'acc': 'accuracy',
           'prec': 'precision_macro',
           'rec': 'recall_macro',
           'f1': 'f1_macro',
           'b_acc': 'balanced_accuracy',
           'auc': 'roc_auc',
           'tpr': tpr_scorer,
           'fpr': fpr_scorer
          }

# Validación cruzada
scores = cross_validate(model,X,y,verbose=2,scoring=scoring, cv=5)

# Nos importan mas los promedios
results = pd.DataFrame(
    data={
        'TIME': scores['fit_time'],
        'ACCURACY': scores['test_acc'],
        'PRECISION': scores['test_prec'],
        'RECALL': scores['test_rec'],
        'F1': scores['test_f1'],
        'B_ACCURACY': scores['test_b_acc'],
        'AUC': scores['test_auc'],
        'TPR': scores['test_tpr'],
        'FPR': scores['test_fpr'],
    }
).mean()
print(model)
print(results)
```

El resultado de correr este último script era una lista con el promedio de los estimadores mencionados anteriormente:

```
LogisticRegression(C=3,
max_iter=1000, penalty='none')
TIME                7.158566
ACCURACY            0.988537
PRECISION           0.964976
RECALL              0.971534
F1                  0.967702
B_ACCURACY          0.971534
AUC                 0.996904
TPR                 0.950502
```

4. Modelo de cascada

Una vez definidos todos los modelos (con los mejores hiperparametros) que se iban a usar, era hora de armar el modelo de cascada final.

La idea es construir clasificadores eficientes en el rechazo de sub-ventanas negativas (que no contienen rostros), es decir, clasificadores para los cuales la tasa de verdaderos negativos sea cercana a uno. Los primeros clasificadores de la cadena, más simples, se utilizan para rechazar la mayoría de las sub-ventanas antes de recurrir a clasificadores más complejos (posteriores en la cadena) para lograr tasas bajas de falsos positivos.

Esto se podría ver con este boceto:

```
def cascada(imagen):
    label1 = modelo1.predict(image)
    if(label1 == true):
        label2 = modelo2.predict(image)
        if(label2 == true):
            label3 = modelo3.predict(image)
            if(label3 == true):
                return true
    return false
```

La idea es que el modelo 1 sea el más simple de todos y que rechace lo antes posible a cualquier imagen que no sea una cara. Todos los modelos pueden rechazar la imagen, pero cuanto más “profundo” se va en la cadena, más complejo es el modelo al que se tiene que enfrentar. Una imagen solo será denominada como “Cara” si todos los modelos afirman que es una cara.

Para lograr esto, definimos todos los modelos seleccionados y los entrenamos con los datos de entrenamiento.

```

# Definir y entrenar modelos

modelRF = RandomForestClassifier(n_estimators=20, max_depth=3, random_state=42)
modelRF.fit(X_train,y_train)

modelRL = LogisticRegression(penalty='l2',C=2, max_iter=1000)
modelRL.fit(X_train,y_train)

modelKNN = KNeighborsClassifier(n_neighbors=20)
modelKNN.fit(X_train,y_train)

modelDT = DecisionTreeClassifier(max_depth=3, random_state=42)
modelDT.fit(X_train,y_train)

modelBG = BaggingClassifier(n_estimators=1, random_state=42)
modelBG.fit(X_train,y_train)

modelGB = GradientBoostingClassifier(n_estimators=1, learning_rate=0.3, max_depth=3, random_state=42)
modelGB.fit(X_train,y_train)

modelAB = AdaBoostClassifier(n_estimators=8, learning_rate=0.5, random_state=42)
modelAB.fit(X_train,y_train)

modelMLP = MLPClassifier(hidden_layer_sizes=(20, 20, 20), max_iter=100, alpha=0.0001, solver='sgd', verbose=10, random_state=21,tol=0.00000001)
modelMLP.fit(X_train,y_train)

```

Luego, definimos nuestra clase Modelo Cascada. Este es justamente el modelo final que toma todos los modelos y los usa como cascada en su función predict. Tuvimos que crear una clase que implemente los métodos fit y predict, para que se pueda comportar efectivamente como si fuera un modelo y luego poder usarlo para un cross-validation por ejemplo.

```

class ModeloDeCascada(BaseEstimator, ClassifierMixin):
    def __init__(self, models) :
        self.models = models
        self.classes_ = [0, 1]

    def fit(self, X, y):
        for model in self.models:
            model.fit(X, y)
        return self

    def predict(self, x):
        predictions = []

        for im in x:
            image_ok = True

            for model in self.models:
                prediction = model.predict([im])[0]
                if(prediction == 0):
                    image_ok = False
                    break
                predictions.append(image_ok)
        return np.array(predictions)

```

Como podemos ver, al modelo de cascada en su inicialización se le pasa la lista de modelos que va a utilizar para su cascada.

En esta parte es de suma importancia el orden en el que le pasamos los modelos. Como dijimos, los modelos más simples deben estar al principio, para poder descartar lo antes posible las imágenes fáciles de detectar que no son caras.

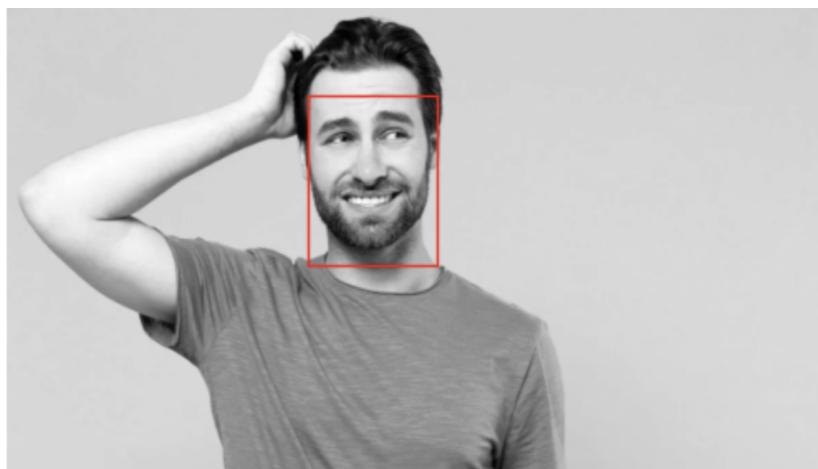
Para confirmar esto, hicimos distintas pruebas cambiando el orden en el que disponemos los modelos seleccionados.

Lo pusimos en una tabla, para poder comparar los tiempos en el cuál, dada la misma imagen y la función principal, se logra detectar la o las caras de la misma.

Tiempo (s)	Accuracy	Precision	1	2	3	4	5	6	7
23.4	0.941	0.999	RF	RL	KNN	DT	AB	GB	BG
11.3	0.941	0.999	GB	RL	DT	KNN	RF	BG	AB
8.8	0.941	0.999	RL	RF	KNN	DT	GB	AB	BG
25	0.941	0.999	RF	RL	KNN	DT	GB	BG	AB
8.7	0.941	0.999	RL	RF	KNN	DT	GB	BG	AB
10.4	0.941	0.999	GB	AB	DT	BG	RF	KNN	RL

Aquí probamos cambiando con varios órdenes, y se dio lo que esperábamos, cuando los primeros modelos van adelante, el procesamiento es más rápido.

Esta fue la imagen con la que se hicieron todas estas pruebas. Todas dan en ese mismo lugar, lo único que cambió fue la velocidad de procesamiento decidiendo frame por frame si es una cara o no.



```
cascadeModel = ModeloDeCascada(models=[modelRL, modelRF, modelKNN, modelDT, modelGB, modelBG, modelAB])

y_pred = cross_val_predict(cascadeModel, X_train, y_train, cv=5)
accuracy = accuracy_score(y_train, y_pred)
print("Average Accuracy:", accuracy)

precision = precision_score(y_train, y_pred)
print("Average Precision:", precision)
```

5. Función principal

Esta es la función “main” a la cuál, dada una imagen cualquiera, imprime un recuadro rojo en el lugar donde detecta una cara.

La función está dividida en 4 partes:

1. Inicialización:

Lo primero que se hace es el tratamiento inicial de las imágenes. Lo primero que hacemos es pasarla a escala de grises. Luego, hacemos uso de nuestra función preprocesamientoImg(img).

```
def preprocesamientoImg(img):
    maxResolucionPosible = 1000 #defino la resolución máxima posible de la foto. Si es mayor, la reduzco
    scales = [4,2,1,0.5] #defino los factores de escala para pasar con la sliding window
    istep = 4 #defino el paso de desplazamiento en la dirección i (verticalmente)
    jstep = 4 #defino el paso de desplazamiento en la dirección j (horizontalmente)

    resolution = max(img.shape)
    if(resolution > 2 * maxResolucionPosible):
        resizer = 0.3
    elif(resolucion > maxResolucionPosible):
        resizer = 0.5
    else:
        resizer = 1

    return resizer, scales, istep, jstep
```

Esta función se encarga de definir algunos parámetros para utilizar en la función. Estos se ponen “a mano” y son los que definen si el procesamiento será más lento y costoso, pero más preciso, o más rápido y menos preciso.

Los datos que se inicializan son:

- **Máxima resolución posible:** define la máxima resolución que puede tener una foto para ser procesada. Si la resolución es mayor, entonces el algoritmo la reescalara. Cuanto mejor sea la resolución, más lento pero más preciso.
- **Escalas de la ventana deslizante:** Aquí definimos los tamaños que tendrán nuestras ventanas deslizantes.
- **Istep y Jstep:** definimos el paso de desplazamiento en i y en j que tendrán las ventanas deslizantes.

Todos los valores que elegimos fueron corroborados para ser los óptimos para el promedio de las imágenes testeadas.

2. Iteración:

A través de un triple for es que se va a procesar toda la imagen.

El primero itera en torno a las escalas para la ventana deslizante. Se define la escala de la misma y respecto a ella el tamaño de los pasos a utilizar.

Una vez definido el tamaño de la ventana, se empieza a recorrer (En el eje x e y) con esa ventana y para cada iteración se extrae un patch. El patch es la porción de la imagen que tiene

capturada la ventana deslizante. Se toman las coordenadas y dimensiones del patch, y se procesa.

3. Procesamiento del patch:

Dentro del procesamiento del patch lo primero que hacemos es extraer las 81 HOG features de las que hablamos antes y desplegarlas en un array al que llamamos: patch_hog. Ese array es el que se usa para pasarle a nuestro modelo de cascada para que prediga si es una cara o no. En caso de que el procesamiento del patch devuelva “true”, entonces las coordenadas, dimensiones y escala de este patch se almacena en el array de detecciones y se sigue con la iteración. Si se retorna que no se encontró ninguna cara, entonces únicamente se sigue con la iteración, para procesar el próximo patch.

4. Supresión:

Una vez terminada la triple iteración, nos quedaremos con un array de detecciones, compuesto por cada patch que el modelo afirmó ser una cara.

A esa lista se le pasa la función de supresión, para quedarnos con una lista con el/los rectángulos suprimidos.

Esta es la lista final que luego se imprime cada detección en la foto.

A continuación, mostramos nuestra función principal “main”

```

def Main(img):
    img = color.rgb2gray(img)
    patch_size=(62,47)
    resizor, scales, istep, jstep = preprocesamientoImg(img)
    if(resizor != 1):
        img = rescale(img, resizor)
        print("La imagen fue reducida a " + str(resizor) + " de su tamaño original")
    detecciones = []
    detecciones_suprimidas = []
    cantDeteccionesPorEscala = 0
    for scale in scales:
        Ni, Nj = (int(scale * s) for s in patch_size)

        for i in range(0, img.shape[0] - Ni, istep):
            for j in range(0, img.shape[1] - Nj, jstep):
                patch = img[i:i + Ni, j:j + Nj]
                if scale != 1:
                    patch = resize(patch, patch_size)

                if(ProcessarParche(patch)):
                    detecciones.append((i, j, Ni, Nj, scale))

        print("Cant. detecciones en escala " + str(scale) + " = " + str(len(detecciones) - cantDeteccionesPorEscala))
        cantDeteccionesPorEscala = len(detecciones)

    detecciones_suprimidas = non_max_suppression1(detecciones, 0.01)
    fig, ax = plt.subplots()
    ax.imshow(img, cmap='gray')
    ax.axis('off')
    print(" ")
    print ("Cantidad total de detecciones: " + str(len(detecciones)))
    print("Cantidad de rostros encontrados: " + str(len(detecciones_suprimidas)))
    for deteccion in detecciones_suprimidas:
        i, j, Ni, Nj, scale = deteccion
        ax.add_patch(plt.Rectangle((j, i), Nj, Ni, edgecolor='red', alpha=1, lw=1, facecolor='none'))
    plt.show()

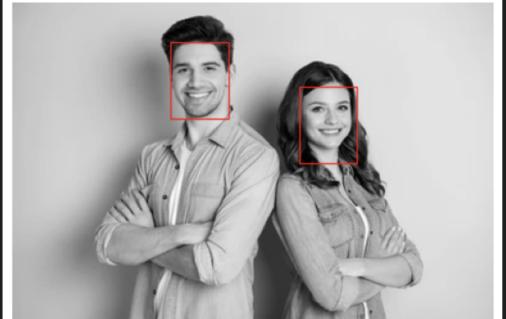
```

Algunos resultados:

```
test_image = plt.imread('cara2.jpg')
Main(test_image)
✓ 3.8s

Cant. detecciones en escala 4 = 0
Cant. detecciones en escala 2 = 0
Cant. detecciones en escala 1 = 5
Cant. detecciones en escala 0.5 = 0

Cantidad total de detecciones: 5
Cantidad de rostros encontrados: 2


```

```
test_image = plt.imread('cara3.jpg')
Main(test_image)
✓ 8.2s

La imagen fue reducida a 0.5 de su tamaño original
Cant. detecciones en escala 4 = 0
Cant. detecciones en escala 2 = 10
Cant. detecciones en escala 1 = 0
Cant. detecciones en escala 0.5 = 1

Cantidad total de detecciones: 11
Cantidad de rostros encontrados: 1


```

```
test_image = plt.imread('cara7.jpg')
Main(test_image)
✓ 18.5s

La imagen fue reducida a 0.3 de su tamaño original
Cant. detecciones en escala 4 = 0
Cant. detecciones en escala 2 = 0
Cant. detecciones en escala 1 = 9
Cant. detecciones en escala 0.5 = 2

Cantidad total de detecciones: 11
Cantidad de rostros encontrados: 7


```

```
test_image = plt.imread('cara9.jpg')
Main(test_image)
✓ 3.7s

Cant. detecciones en escala 4 = 0
Cant. detecciones en escala 2 = 0
Cant. detecciones en escala 1 = 3
Cant. detecciones en escala 0.5 = 1

Cantidad total de detecciones: 4
Cantidad de rostros encontrados: 3


```

```
test_image = plt.imread('cara6.jpg')
Main(test_image)
✓ 2m 17.5s

Cant. detecciones en escala 4 = 0
Cant. detecciones en escala 2 = 0
Cant. detecciones en escala 1 = 17
Cant. detecciones en escala 0.5 = 97

Cantidad total de detecciones: 114
Cantidad de rostros encontrados: 21


```

```
test_image = plt.imread('cara1.jpg')
Main(test_image)
✓ 9.0s

La imagen fue reducida a 0.5 de su tamaño original
Cant. detecciones en escala 4 = 0
Cant. detecciones en escala 2 = 17
Cant. detecciones en escala 1 = 0
Cant. detecciones en escala 0.5 = 0

Cantidad total de detecciones: 17
Cantidad de rostros encontrados: 1


```

Veamos qué pasa si cambiamos los steps y las escalas:

Caso 1:

Steps = 4px

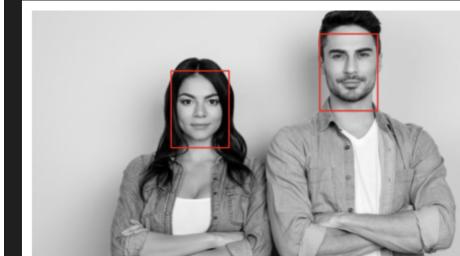
Escalas = [4, 2, 1, 0.5]

Tiempo total = 15.2s

Cantidad total de detecciones = 22

```
test_image = plt.imread('cara4.jpg')
Main(test_image)
✓ 15.2s
Cant. detecciones en escala 4 = 0
Cant. detecciones en escala 2 = 21
Cant. detecciones en escala 1 = 0
Cant. detecciones en escala 0.5 = 1

Cantidad total de detecciones: 22
Cantidad de rostros encontrados: 2
```



Caso 2:

Steps = 2px

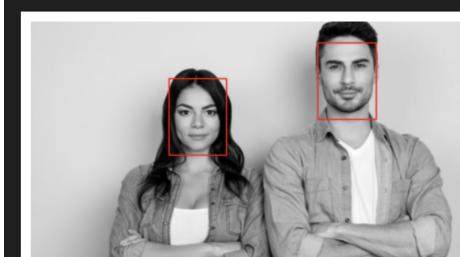
Escalas = [4, 2, 1, 0.5]

Tiempo total = 1.05m

Cantidad total de detecciones = 79

```
test_image = plt.imread('cara4.jpg')
Main(test_image)
✓ 1m 0.5s
Cant. detecciones en escala 4 = 0
Cant. detecciones en escala 2 = 76
Cant. detecciones en escala 1 = 1
Cant. detecciones en escala 0.5 = 2

Cantidad total de detecciones: 79
Cantidad de rostros encontrados: 2
```



Caso 3:

Steps = 2px

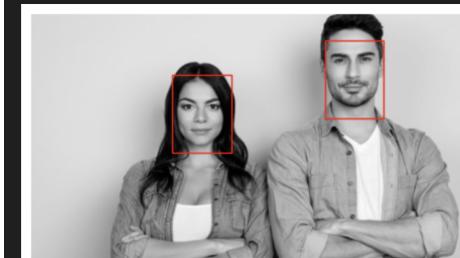
Escalas = [4,3,2,1.5,1,0.5]

Tiempo total = 1.40m

Cantidad total de detecciones = 126

```
test_image = plt.imread('cara4.jpg')
Main(test_image)
✓ 1m 40.9s
Cant. detecciones en escala 4 = 0
Cant. detecciones en escala 3 = 0
Cant. detecciones en escala 2 = 76
Cant. detecciones en escala 1.5 = 47
Cant. detecciones en escala 1 = 1
Cant. detecciones en escala 0.5 = 2

Cantidad total de detecciones: 126
Cantidad de rostros encontrados: 2
```



Esto es un ejemplo muy claro en cuanto al trade off que hay que hacer constantemente entre la exactitud y la velocidad. Vemos como en el primer caso de todos, tenemos un modelo muy rápido y bastante preciso en cuanto a ubicar las caras. Encontró un total de 22 caras, suprimidas en 2.

En el otro extremo, también tenemos otro modelo que encontró 126 detecciones, pero en casi 2 minutos. Probablemente sea más exacto, pero al final de cuentas, el primero hizo casi lo mismo en un tiempo 8 veces menor.