Práctica Extracción de Rasgos

Extracción de Características en Imágenes

Universidad de Granada

Juan Ignacio Isern Ghosn
Correo: jisern@correo.ugr.es

February 27, 2019

Contents

1 Resumen de la práctica

El documento que a continuación se presenta se corresponde con la primera práctica de la asignatura Extracción de Características en Imágenes, del máster universitario en Ciencia de Datos e Ingeniería de Computadores. El objetivo de esta práctica de evaluación es extraer diferentes tipos de rasgos de una imagen y usarlos para aprender conceptos sencillos mediante el uso de clasificadores.

1.1 Contenido de la práctica

Esta práctica se compone de una primera parte **obligatoria** y una segunda **voluntaria**:

- La **parte obligatoria** se conforma por la evaluación del modelo de clasificación SVM con HoG (Histogram of Gradients) visto en clase, ayudándonos de medidas de bondad y mediante el ajuste de parámetros. A su vez, esta parte incluye la implementación del descriptor LBP (Local Binary Pattern) sobre el cual deberemos realizar una evaluación del modelo resultante similar a la anterior.
- La **parte voluntaria** se compone de la implementación del descriptor LBP Uniforme y su evaluación con medidas de bondad y ajuste de parámetros, la combinación en el uso de descriptores sobre los datos y su evaluación y por último, la implementación de una búsqueda e identificación de personas haciendo uso de dichos descriptores, sobre imágenes de cualquier tamaño

1.2 Conjunto de datos

El conjunto de datos se trata de una base de datos de 5406 imágenes (2416 de peatones y 2990 de fondo), proporcionada por el personal docente de la asignatura. Cada una de estas imágenes tienen un tamaño de 124x64 px.

1.3 Lenguaje de programación y entorno de desarrollo empleado

El lenguaje de programación empleado para este trabajo ha sido *Python*, en su versión 2.7. Del mismo modo, el entorno de desarrollo utilizado ha sido Pycharm en su versión *community*, de la compañía de software *Jetbrains*.

1.4 Código resultante e instrucciones para su ejecución

El código resultante al proyecto que se describe en este informe se puede encontrar en la carpeta *src_ExtraccionRasgos*, conjuntamente entregada con este fichero. Para ejecutar el proyecto es necesario instalar las dependencias indicadas en los *import* de cada uno de los ficheros. Del mismo modo, es necesario incluir las imágenes de entrenamiento y test dentro de las carpetas *train* y *test* del directorio *data*, a fin de entrenar cada uno de los clasificadores empleados durante el proceso de la práctica.

Aunque se ha generado, no ha sido posible entregar un ejecutable del código generado, pues se ha de tener en cuenta el intérprete de Python y el compilador de C del usuario que ejecuta el código. La solución es la de crear un fichero que internamente tenga un compilador e intérprete, si bien el tamaño del mismo hace inasumible su entrega por los límites de espacio de la plataforma *Prado*.

2 Parte obligatoria

A continuación, se detallan las tareas llevadas a cabo para la correcta realización de la primera parte obligatoria del trabajo.

2.1 Evaluación clasificación HoG+SVM: medidas de bondad y parámetros

En el ejercicio de clase se entrenó un clasificador SVM con el descriptor HOG y se usó para predecir la clase (persona vs. fondo) de una imagen dada. No obstante, no se obtuvieron medidas de la bondad de la clasificación usando el conjunto de imágenes test, ni tampoco se barajaron diferentes particiones entre datos de entrenamiento y además, se usaron los parámetros básicos en el SVM (con un kernel lineal).

2.1.1 HOG + SVM: Medidas de bondad del clasificador generado

Para calcular el rendimiento del clasificador visto en clase (con núcleo lineal y parámetro C = 1) se utilizan cuatro medidas distintas: Exactitud, precisión, sensibilidad y F1-score. Para ello se hace uso del paquete de medidas presente en la librería *Scikit-learn*. El código que implementa dicha tarea (función **std_clf_metrics**) se muestra a continuación (Anexo: ***TestBasico.py***):

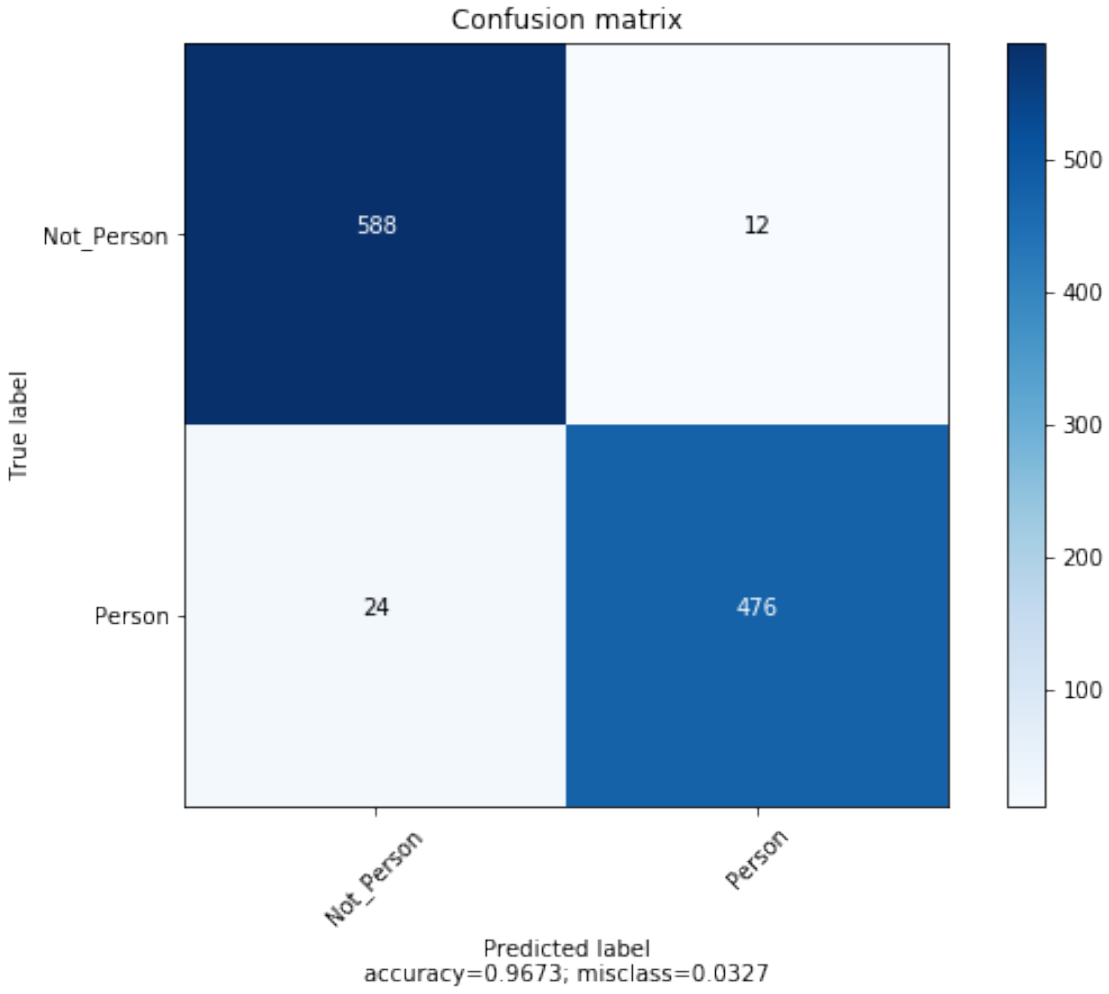
```
def std_clf_metrics(classes_test, prediction, save=False, name=None):
    """Compute metrics for a given prediction.

    Compute Accuracy, precission, recall, F1 and print confusion
    matrix for given prediction.

    Args:
        (int[]) classes_test: Real label of the data
        (int[]) prediction: Predicted label of the data
        (bool) save: Save CV score.
        (String) name: name of score file.
    """
    scores = {"Exactitud": [metrics.accuracy_score(classes_test, prediction)],
              "Precision": [metrics.precision_score(classes_test, prediction)],
              "Sensibilidad": [metrics.recall_score(classes_test, prediction)],
              "F1-Score": [metrics.f1_score(classes_test, prediction)]}
    df = pd.DataFrame.from_dict(scores)
    print(df)
    cm = (metrics.confusion_matrix(classes_test, prediction))
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        plot_confusion_matrix(cm, normalize=False)
    if save:
        write_data(df, "../scores/" + name + ".pkl")
        write_data(cm, "../scores/" + name + "_cm.pkl")
```

Tanto el resultado de la ejecución de la función anterior sobre las predicciones obtenidas del clasificador entrenado para HOG, como la matriz se muestra a continuación:

Exactitud	F1-Score	Precision	Sensibilidad
0.967273	0.963563	0.97541	0.952



Tal y como dejan entrever los resultados, la clasificacion efectuada sobre el test es bastante buena. Podemos observar como la exactitud es alta, siendo una buena medida de rendimiento en este caso, que están bien balanceados los datos de cada una de las clases. Sin embargo, se puede apreciar en la precisión y sensibilidad como se clasifica algo mejor la clase mayoritaria (fondos) frente a la minoritaria (personas), si bien esta diferencia no es tan significativa (en torno al 1%).

2.1.2 HOG + SVM: Validación cruzada

Haciendo uso de la librería *Scikit-learn* y de su función *cross_validate*, se lleva a cabo un 10-fold CV para este primer clasificador con núcleo lineal y parámetros por defecto (C = 1). El código que implementa dicha tarea (función **cv_standard_svm**) se muestra a continuación (Anexo: ***TestBasico.py***):

```
def cv_standard_svm(data, classes, save=True, name=None, cv = 10):
    """Compute CV with default parameters.

    Compute cross-validation for standard linear SVM classifier

    Args:
        (float[][]) data: Image descriptors.
        (int[]) classes: Label of the data
        (bool) save: Save CV score.
        (String) name: name of score file.
        (int) cv: Number of folds
    Returns:
        DataFrame: Scores for each CV split.
    """
    clf = svm.SVC(kernel='linear')
    scoring = ['accuracy', 'precision', 'recall', 'f1']
    scores = cross_validate(clf, data, classes, cv=cv, n_jobs=-1,
                            scoring=scoring, verbose=10, return_train_score=False)
    df = pd.DataFrame.from_dict(scores)
    if save: write_data(df, '../scores/' + name + ".pkl")
    return df
```

El resultado de la ejecución del 10-fold CV con SVM de núcleo lineal y parámetro C = 1 sobre los datos de los descriptores HOG de las imágenes tienen como resultados los siguientes:

	fit_time	score_time	test_accuracy	test_f1	test_precision	test_recall
0	30.529023	10.676842	0.974122	0.970711	0.983051	0.958678
1	29.674839	10.461919	0.948244	0.941667	0.949580	0.933884
2	30.879635	10.828638	0.963031	0.958678	0.958678	0.958678
3	29.616469	10.345715	0.951941	0.945378	0.961538	0.929752
4	28.516598	10.614437	0.966728	0.962343	0.974576	0.950413
5	28.390929	10.631926	0.940850	0.933884	0.933884	0.933884
6	28.103214	10.390336	0.953704	0.948665	0.939024	0.958506
7	28.208798	10.455860	0.951852	0.946058	0.946058	0.946058
8	24.560487	9.038676	0.962963	0.957806	0.974249	0.941909
9	24.549541	9.231040	0.970370	0.966527	0.974684	0.958506

Como se puede apreciar, los resultados entre los distintos splits del CV no varían demasiado (+-3%) y son de un orden parecido al mostrado en el apartado anterior. Esto nos hace pensar que el clasificador parece que no generaliza mal del todo. A continuación, se muestra la media de los resultados:

	fit_time	score_time	test_accuracy	test_f1	test_precision	test_recall
9	28.302953	10.267539	0.958381	0.953172	0.959532	0.947027

2.1.3 HOG + SVM: Optimización de parámetros

Tal y como se ha mostrado en los apartados anteriores, los clasificadores entrenados tienen núcleo lineal y parámetros por defecto. A fin de buscar aquellos parámetros que puedan incrementar el rendimiento del SVM se efectúa una comparativa entre distintos clasificadores, cada uno de ellos con distinto núcleo y parámetros. El código que implementa dicha tarea (función **find_best_params**) se muestra a continuación (Anexo: ***TestBasico.py***):

```
def find_best_params(data, classes, save=True, name=None):
        """SVM parameter tunning.

        Search best kernel and hyperparameter.

        Args:
            (float[][]) data: Image descriptors.
            (int[]) classes: Label of the data
            (bool) save: Save CV score.
            (String) name: name of score file.
        Returns:
            DataFrame: Scores for kernel & parameter combination.
        """
        tuned_parameters = [{'kernel': ['rbf'], 'gamma': [0.001, 0.01, 0.1],
                             'C': [1, 10, 100]},
                            {'kernel': ['sigmoid'], 'gamma': [0.001, 0.01, 0.1],
                             'C': [1, 10, 100]},
                            {'kernel': ['linear'], 'C': [1, 10, 100]}]
        svc = svm.SVC()
        scoring = ['accuracy', 'precision', 'recall', 'f1']
        clf = GridSearchCV(svc, tuned_parameters, cv=5, verbose=10, n_jobs=-1,
                           return_train_score=False, scoring=scoring,
                           refit='accuracy')
        res = clf.fit(data, classes)
        df = pd.DataFrame.from_dict(res.cv_results_).iloc[:, 2:13]
        print df
        if save: write_data(df, '../scores/' + name + ".pkl")
```

Tal y como se puede apreciar en la implementación de la función anterior, se realiza una búsqueda del mejor kernel de entre tres: Lineal, Sigmoidal y Gaussiano. A su vez, se establecen distintos valores de parámetros para el C y gamma, siendo este último propio del núcleo Sigmoidal y Gaussiano. Los resultados de esta búsqueda de parámetros se muestran a continuación:

	mean_test_accuracy	mean_test_f1	mean_test_precision	mean_test_recall
9	0.977987	0.975137	0.984422	0.966059
12	0.977987	0.975137	0.984422	0.966059
6	0.977617	0.974719	0.984000	0.965645
10	0.974658	0.971277	0.984811	0.958197
3	0.973548	0.969848	0.988433	0.951988

```
25            0.972438      0.968616            0.986420          0.951575
13            0.972253      0.968583            0.980555          0.956955
7             0.971328      0.967158            0.990104          0.945366
14            0.969108      0.964655            0.986298          0.944125
28            0.968553      0.964350            0.978144          0.951162
11            0.964484      0.959394            0.980682          0.939157
22            0.964299      0.959192            0.980270          0.939157
29            0.964299      0.959192            0.980270          0.939157
30            0.964299      0.959192            0.980270          0.939157
31            0.963744      0.958989            0.970062          0.948263

   param_C param_gamma param_kernel
9       50        0.01          rbf
12     100        0.01          rbf
6       10        0.01          rbf
10      50       0.001          rbf
3        1        0.01          rbf
25      50       0.001      sigmoid
13     100       0.001          rbf
7       10       0.001          rbf
14     100      0.0001          rbf
28     100       0.001      sigmoid
11      50      0.0001          rbf
22      10       0.001      sigmoid
29     100      0.0001      sigmoid
30    0.01         NaN       linear
31       1         NaN       linear
```

Los resultados arriba mostrados se ordenan por la exactitud de la predicción. Tal y como se puede apreciar, el kernel Gaussiano consigue en término generales el mejor resultado, seguido del Sigmoidal y siendo el Lineal el peor de los tres. Los mejores resultados de los obtenidos son aquellos que usan una gamma de 0.01 y un C tanto de 50 como de 100.

2.2 Evaluación clasificación LBP+SVM: medidas de bondad y parámetros

A fin de evaluar el rendimiento del descriptor LBP (Local Binary Pattern) sobre la clasificación de imágenes de personas vs fondos, se lleva a cabo la implementación de dicho descriptor y su posterior puesta en práctica por medio del mismo proceso en el caso anterior del HoG:

2.2.1 Implementación descriptor LBP:

Se ha llevado a cabo la implementación de la mayoría del trabajo de esta práctica por medio del lenguaje de programación interpretado Python, en su versión 2.7. Por ello, la implementación de este descriptor no ha sido diferente. A continuación se muestra la clase generada que implementa la instancia de dicho descriptor (Anexo: ***LBPDescriptor.py***):

```
import cv2 as cv2
import _functions
```

```
class LBPDescriptor:

    #Local Binary Pattern descriptor object class

    def compute(self, img):
        """Compute descriptor.

        Args:
            img (int[][][]): An RGB image.
        Returns:
            float[]: Local Binary Pattern descriptor.
        """
        grey_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        feat = _functions.lbp(grey_img)
        return feat
```

Tal y como se puede observar, el cálculo del valor del descriptor apunta a una función *lbp*, del fichero *_functions.pyx. *Esto es así debido a que los cálculos concretos que debe realizar el descriptor están escritos en Cython, lenguaje que permite transformar Python a código compilado en C, lo cual incrementa la velocidad de su ejecución. El código de la función* lbp* como de las auxiliares de las cuales se ayuda se muestran a continuación (Anexo: **_functions.pyx**):

```
def lbp(unsigned char[:,:] img):
    """Local Binary Pattern calculation.

    Args:
        (unsigned char[:,:]) img: Greyscale image.
    Returns:
        (double[:]) histograms: Histograms of image blocks.
    """
    cdef int y_lim = img.shape[0]
    cdef int x_lim = img.shape[1]
    cdef double[:,:] texture_map = get_texture_map(img)

    hists = []

    for y in range(0, y_lim - 8, 8):
        for x in range(0, x_lim - 8, 8):
            hist = compute_block_lbp(texture_map, y, x)
            hists.append(hist/np.linalg.norm(hist))

    return np.concatenate(hists)

def get_texture_map(unsigned char[:,:] img):
    """LBP texture map calculation.

    Args:
```

```
        (unsigned char[:,:]) img: Greyscale image.
    Returns:
        (double[:,:]) texture_map: Texture map with LBP.
    """
    cdef int y_lim = img.shape[0]
    cdef int x_lim = img.shape[1]
    cdef double[:,:] texture_map = np.zeros((y_lim, x_lim))
    cdef int val = 0
    cdef int pt = 0

    for y in range(1, y_lim - 1):
        for x in range(1, x_lim - 1):
            val = img[y, x]
            pt = 0
            # Bit shifting and or operation for add 2^N to the pattern.
            pt = pt | (1 << 7) if val <= img[y - 1, x - 1] else pt
            pt = pt | (1 << 6) if val <= img[y - 1, x] else pt
            pt = pt | (1 << 5) if val <= img[y - 1, x + 1] else pt
            pt = pt | (1 << 4) if val <= img[y, x + 1] else pt
            pt = pt | (1 << 3) if val <= img[y + 1, x + 1] else pt
            pt = pt | (1 << 2) if val <= img[y + 1, x] else pt
            pt = pt | (1 << 1) if val <= img[y + 1, x - 1] else pt
            pt = pt | (1 << 0) if val <= img[y, x - 1] else pt
            texture_map[y, x] = pt

    # Copy first and last rows/columns in edge rows/columns:
    texture_map[0,:] = texture_map[1, :]
    texture_map[texture_map.shape[0]-1,:] = texture_map[texture_map.shape[0]-2,:]
    texture_map[:,0] = texture_map[:, 1]
    texture_map[:,texture_map.shape[1]-1] = texture_map[:,texture_map.shape[1]-2]

    return texture_map


def compute_block_lbp(double[:,:] texture_map, int i_start, int j_start):
    """Local Binary Pattern image block calculation.

    Args:.
        (int) i_start: Block x start index.
        (int) j_start: Block y start index.
        (double[:,:]) texture_map: Texture map with LBP.
    Returns:
        (double[:]) histogram: Histogram for each image block.
    """
    cdef double[:] hist = np.zeros(256)
    cdef double val = 0

    for i in range(i_start, i_start + 16) :
```

```
            for j in range(j_start, j_start + 16) :
                val = texture_map[i,j]
                hist[(<int>val)] = hist[(<int>val)] + 1

        return hist
```

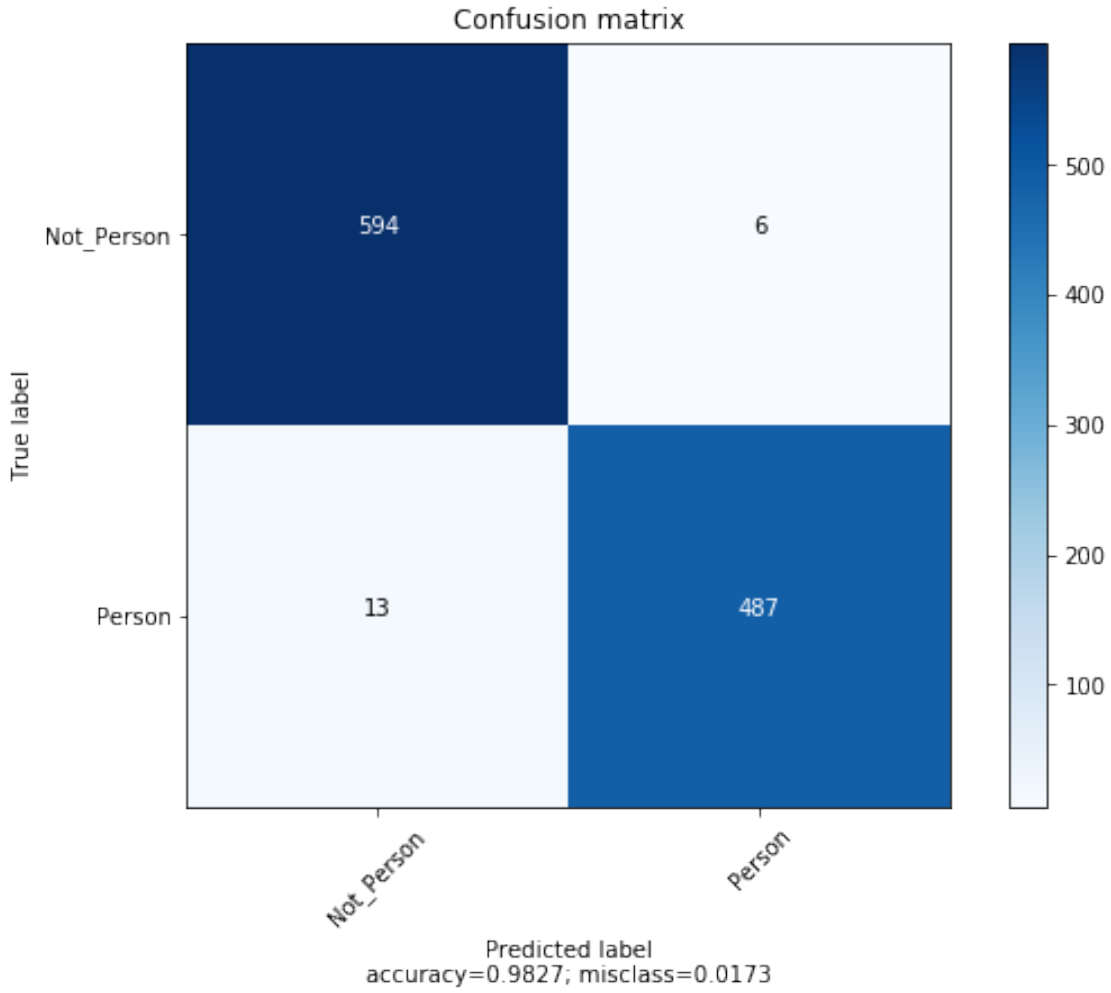
Tal y como se puede observar en las funciones anteriores el procedimiento es relativamente sencillo:

1. En primer lugar, se calcula el mapa de texturas correspondiente a toda la imagen, añadiendo en la posición del bit de la imagen que se examina en cada momento el valor de acuerdo al patrón obtenido de los 8 vecinos.
2. A continuación se calcula el histograma de cada uno de los bloques del mapa de texturas anterior. La concatenación de todos estos bloques serán el descriptor LBP para cada una de las imágenes.

2.2.2 LBP + SVM: Medidas de bondad del clasificador generado

Tal y como hicimos con el análisis del clasificador SVM generado en clase con HOG, realizamos un análisis similar del rendimiento del mismo clasificador, ahora entrenado con el descriptor LBP generado. Tal y como ya se comentó anteriormente, la función que implementa esta tarea es **cv_standard_svm** (Anexo: ***TestBasico.py***). Los resultados que se obtienen para el conjunto por defecto de train y test dado para la práctica será el siguiente:

	Exactitud	F1-Score	Precision	Sensibilidad
0	0.982727	0.980866	0.98783	0.974



Desde un primer momento caemos en la cuenta de que los resultados son mejores a los obtenidos con el HOG. Supera casi en 1.5% más en todas las medidas al clasificador entrenado con descriptores HOG. Del mismo modo que este, también tiene como mayor dificultad la correcta clasificación de personas, si bien la sensibilidad se ha incrementado en la misma medida que el resto de las medidas. La matriz de confusión nos permite visualizar fácilmente y confirmar todas estas intuiciones dadas por las medidas.

2.2.3 LBP + SVM: Validación cruzada

Haciendo uso de la librería *Scikit-learn* y de su función *cross_validate*, tal y como haríamos para el epígrafe correspondiente al HOG, se lleva a cabo un 10-fold CV para el clasificador con núcleo lineal y parámetros por defecto (C = 1). Tal y como ya se comentó anteriormente, la función que implementa esta tarea es **cv_standard_svm** (Anexo: ***TestBasico.py***). Los resultados obtenidos para cada uno de los splits del CV se muestran a continuación:

	fit_time	score_time	test_accuracy	test_f1	test_precision	test_recall
0	229.6107	68.155271	0.992606	0.991770	0.987705	0.995868
1	232.8632	72.249491	0.985213	0.983471	0.983471	0.983471
2	224.0703	66.744454	0.988909	0.987654	0.983607	0.991736
3	219.5369	66.497677	0.988909	0.987654	0.983607	0.991736
4	202.3894	65.349396	0.994455	0.993789	0.995851	0.991736
5	203.0431	67.512402	0.990758	0.989733	0.983673	0.995868
6	209.6558	69.569818	0.981481	0.978992	0.991489	0.966805
7	195.1940	68.361638	0.981481	0.979339	0.975309	0.983402
8	159.1803	56.052726	0.983333	0.981132	0.991525	0.970954
9	155.8142	57.944455	0.990741	0.989648	0.987603	0.991701

Del mismo modo, los resultados medios de la ejecución del CV se muestran a continuación:

fit_time	score_time	test_accuracy	test_f1	test_precision	test_recall
203.1358	65.843733	0.987789	0.986318	0.986384	986328

En primer lugar, llama la atención de estos resultados el incremento del tiempo entrenamiento del clasificador con respecto al HOG. Esto es debido a la alta dimensionalidad de los datos, pues LBP genera 26.880 características para cada una de nuestras imágenes. En cuanto a la bondad del ajuste, se han incrementado de media todas las medidas en un 3% respecto al clasificador con HOG. Al contrario que sucedía anteriormente, la sensibilidad del clasificador está a la par de la precisión, por lo que podemos intuir que está clasificando en mejor medida a las personas y lo hace con una efectividad al menos parecida a los fondos.

2.2.4 LBP + SVM: Optimización de parámetros

En este epígrafe buscaremos aquellos parámetros que mejores resultados dan para el SVM entrenado con el descriptor LBP de las imágenes.Así se efectúa una comparativa entre distintos clasificadores, cada uno de ellos con distinto núcleo y parámetros. La función que implementa dicha tarea se ha mostrado anteriormente en el caso del descriptor HOG (**find_best_params**, Anexo: ***TestBasico.py***). Así, de la búsqueda de parámetros efectuada, se muestran aquellos con mejores resultados:

	mean_test_accuracy	mean_test_f1	mean_test_precision	mean_test_recall	\
4	0.991491	0.990494	0.988883	0.992136	
7	0.991491	0.990494	0.988883	0.992136	
6	0.989271	0.987990	0.988007	0.987996	
1	0.988901	0.987602	0.985984	0.989239	
20	0.988161	0.986750	0.986357	0.987168	
19	0.988161	0.986750	0.986357	0.987168	
18	0.988161	0.986750	0.986357	0.987168	
3	0.987976	0.986557	0.985963	0.987169	
15	0.987976	0.986547	0.985952	0.987168	
12	0.985572	0.983884	0.982692	0.985100	
10	0.978172	0.975616	0.974427	0.976822	

```
0            0.974288      0.971458            0.964159          0.978892
13           0.969108      0.965591            0.961494          0.969784
9            0.966889      0.963418            0.951569          0.975581
16           0.950610      0.944442            0.949446          0.939567

   param_C param_gamma param_kernel
4       10        0.01          rbf
7      100        0.01          rbf
6      100       0.001          rbf
1        1        0.01          rbf
20     100         NaN       linear
19      10         NaN       linear
18       1         NaN       linear
3       10       0.001          rbf
15     100       0.001      sigmoid
12      10       0.001      sigmoid
10       1        0.01      sigmoid
0        1       0.001          rbf
13      10        0.01      sigmoid
9        1       0.001      sigmoid
16     100        0.01      sigmoid
```

Al igual que ocurriese con el clasificador entrenado con HOG, en este caso resulta también que el mejor kernel es en términos generales el Gaussiano. Los mejores resultados se encuentran por encima del 99%, un resultado superior a los anteriores obtenidos con HOG. Algo interesante que se puede observar acerca del kernel Lineal es, que antes resultaba ser el peor, ahora es una buena alternativa en términos de rendimiento y bondad, pues se sitúa por delante del Sigmoidal.

3 Mejoras voluntarias

A continuación, se detallan las tareas llevadas a cabo para la correcta realización las tareas propuestas como voluntarias.

3.1 LBP-uniforme

En este apartado del informe se muestra la implementación del descriptor LBP uniforme y el análisis realizado para la evaluación de su rendimiento respecto a su implementación no uniforme y respecto al descriptor HOG.

3.1.1 Implementación descriptor LBP-uniforme:

Al igual que su versión no uniforme, se ha llevado a cabo la implementación de este descriptor mediante Python, en su versión 2.7. A continuación se muestra la clase generada que implementa la instancia de dicho descriptor (Anexo: ***UniformLBPDescriptor.py***):

```
import cv2 as cv2
import _functions

class UniformLBPDescriptor:

    """
        Uniform Local Binary Pattern descriptor object class

    Attributes:
        (int[]) uniform_patterns: LBP histogram uniform values.
    """

    uniform_patterns = []

    def __init__(self):
        """Initialize descriptor.

        Initialize descriptor and compute uniform patterns.
        """
        self.uniform_patterns = self.__get_uniform_patterns()

    def compute(self, img):
        """Compute descriptor.

        Args:
            (int[][][]) img: An RGB image.
        Returns:
            float[]: Uniform Local Binary Pattern descriptor.
        """
        grey_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        feat = _functions.uniform_lbp(grey_img, self.uniform_patterns)
```

```
        return feat

    def __get_uniform_patterns(self):
        """Compute uniform patterns.

        Compute uniform patterns (Number of transitions <= 2)
        and update instance values.

        45 ->  00101101:
            -Number of transitions: 6 (Non-uniform)

        255 -> 11111110:
            -Number of transitions: 2 (Uniform)

        Returns:
            int[]: Uniform patterns for descriptor instance.
        """
        patterns = []
        for i in range(2**8):
            binary = map(int, format(i, '08b'))
            first = prev = tran = 0
            for j in range(binary.__len__()):
                if j == 0:
                    first = binary[j]
                    prev = binary[j]
                tran = tran + 1 if binary[j] != prev else tran
                tran = tran + 1 if (j == (binary.__len__() - 1))
                    & (binary[j] != first) else tran
                prev = binary[j]
            patterns.append(i) if tran <= 2 else self.uniform_patterns

        return patterns
```

Tal y como se puede observar, el cálculo del valor del descriptor apunta a una función *uniform_lbp*, del fichero *_functions.pyx*. Esto es así debido a que los cálculos concretos que debe realizar el descriptor están escritos en Cython, lenguaje que permite transformar Python a código compilado en C, lo cual incrementa la velocidad de su ejecución. A esta función se le tienen que pasar aquellos patrones LBP considerados uniformes, que serán calculados en el método **__get_uniform_patterns**. Para el cálculo de estos patrones que son uniformes, se consideran aquellas transiciones entre 0 y 1 que se dan en los patrones, considerando uniforme aquellos que como máximo tienen 2.

Por su parte, el código de la función *uniform_lbp* como de las auxiliares de las cuales se ayuda se muestran a continuación (Anexo: **_functions.pyx**):

```
def uniform_lbp(unsigned char[:,:] img, list uniform_patterns):
    """Uniform Local Binary Pattern calculation.

    Args:
```

```
        (unsigned char[:,:]) img: Greyscale image.
        (list) uniform_patterns: List of considered uniform patterns.
    Returns:
        (double[:]) histograms: Histograms of image blocks.
    """
    cdef int y_lim = img.shape[0]
    cdef int x_lim = img.shape[1]
    cdef double[:,:] texture_map = get_texture_map(img)

    hists = []

    for y in range(0, y_lim - 8, 8):
        for x in range(0, x_lim - 8, 8):
            hist = compute_block_ulbp(texture_map, y, x, uniform_patterns)
            hists.append(hist/np.linalg.norm(hist))

    return np.concatenate(hists)


def get_texture_map(unsigned char[:,:] img):
    ...


def compute_block_ulbp(double[:,:] texture_map, int i_start,
                       int j_start, list uniform_patterns):
    """Uniform Local Binary Pattern image block calculation.

    Args:.
        (int) i_start: Block x start index.
        (int) j_start: Block y start index.
        (double[:,:]) texture_map: Texture map with LBP.
    Returns:
        (double[:]) histogram: Histogram for each image block.
    """
    cdef double[:] hist = np.zeros(59)
    cdef double val = 0

    for i in range(i_start, i_start + 16) :
        for j in range(j_start, j_start + 16) :
            val = texture_map[i, j]
            val = (<int>uniform_patterns.index(val))
            if val in uniform_patterns:
                hist[val] = hist[val] + 1
            else:
                hist[len(uniform_patterns)] += 1
    return hist
```

Al igual que con el LBP no uniforme, el procedimiento se estructura tal y como sigue:

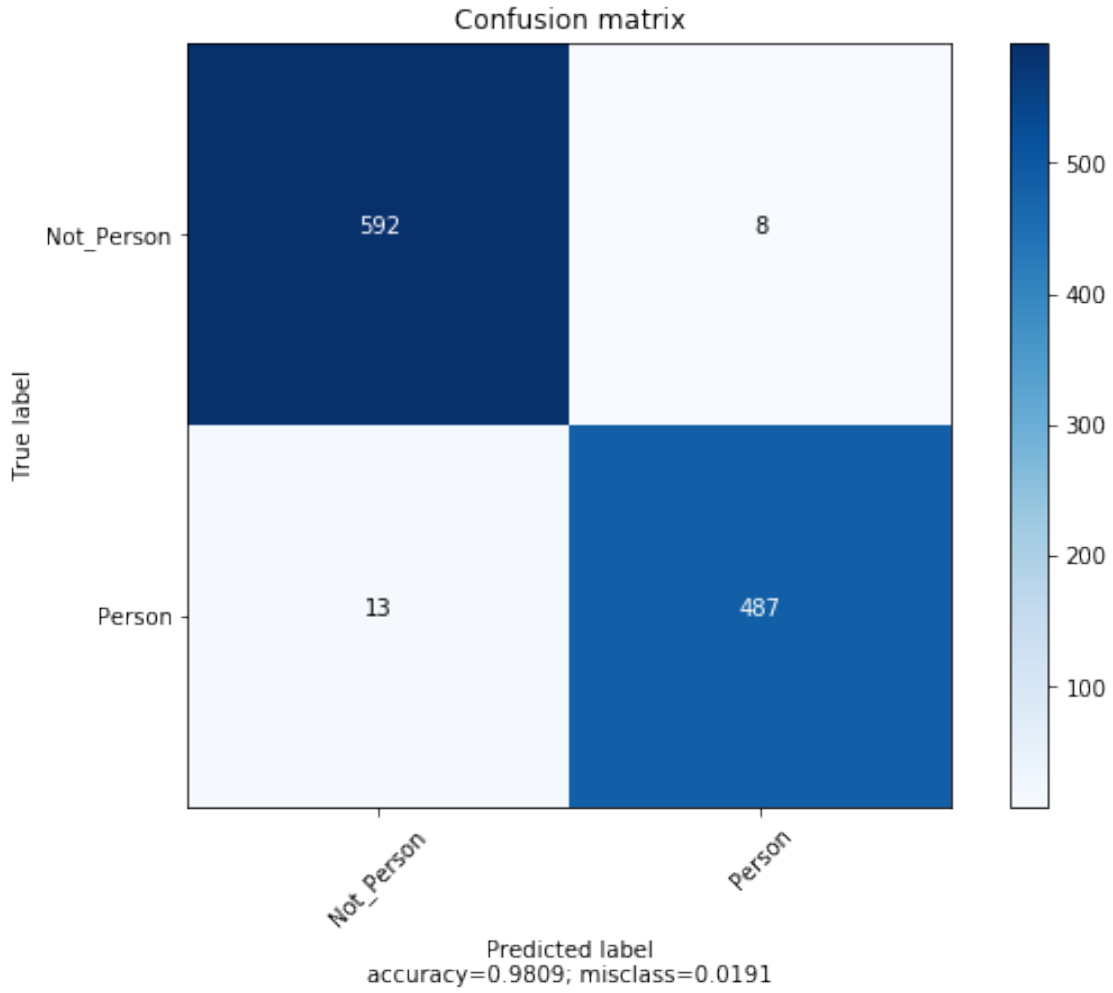
1. En primer lugar, se calcula el mapa de texturas correspondiente a toda la imagen, añadiendo en la posición del bit de la imagen que se examina en cada momento el valor de acuerdo al patrón obtenido de los 8 vecinos.

2. A continuación se calcula el histograma de cada uno de los bloques del mapa de texturas anterior. Cabe remarcar que en este histograma se acumulará la ocurrencia de los valores uniformes, almacenando la de aquellos que no lo sean en la última posición del histograma de cada bloque. La concatenación de todos estos histogramas de bloque serán el descriptor-LBP Uniforme para cada una de las imágenes.

3.1.2 LBP-Uniforme + SVM: Medidas de bondad del clasificador generado

Tal y como hemos hecho tanto para el descriptor HOG como con el LBP, realizamos un análisis similar del rendimiento del clasificador con parámetros por defecto y nucleo Lineal, ahora entrenado con el descriptor LBP-Uniforme implementado. Tal y como ya se comentó anteriormente, la función que implementa esta tarea es **cv_standard_svm** (Anexo: ***TestBasico.py***). Los resultados que se obtienen para el conjunto por defecto de train y test dado para la práctica será el siguiente:

Exactitud	F1-Score	Precision	Sensibilidad
0.980909	0.978894	0.983838	0.974



A simple vista, el resultado para el conjunto de train y test dado es muy bueno, siendo algo inferior que el LBP, hasta ahora el mejor de los tres analizados. Al igual que ocurriera en este clasificador con los dos descriptores anteriores, la sensibilidad parece ser el aspecto más debil.

3.1.3 LBP-Uniforme + SVM: Validación cruzada

Al igual que hemos hecho con los descriptores anteriores y por medio del uso de la librería *Scikit-learn* y de su función *cross_validate*, se lleva a cabo un 10-fold CV para el clasificador con núcleo lineal y parámetros por defecto (C = 1). Tal y como ya se comentó anteriormente, la función que implementa esta tarea es **cv_standard_svm** (Anexo: ***TestBasico.py***). Los resultados obtenidos para cada uno de los splits del CV se muestran a continuación:

	fit_time	score_time	test_accuracy	test_f1	test_precision	test_recall
0	38.284930	15.492903	0.992606	0.991701	0.995833	0.987603
1	37.929144	15.341810	0.988909	0.987603	0.987603	0.987603
2	37.800704	15.316147	0.987061	0.985626	0.979592	0.991736
3	37.589307	15.170854	0.988909	0.987705	0.979675	0.995868

4	40.913787	16.121714	0.988909	0.987654	0.983607	0.991736
5	39.851570	15.849496	0.988909	0.987654	0.983607	0.991736
6	40.874236	15.959079	0.987037	0.985386	0.991597	0.979253
7	38.928760	15.069462	0.977778	0.975104	0.975104	0.975104
8	30.919621	11.332151	0.981481	0.979079	0.987342	0.970954
9	31.299038	11.650667	0.988889	0.987603	0.983539	0.991701

Del mismo modo, los resultados medios de la ejecución del CV se muestran a continuación:

	fit_time	score_time	test_accuracy	test_f1	test_precision	test_recall
9	37.43911	14.730428	0.987049	0.985512	0.98475	0.986329

En primer lugar, el aspecto que más destaca de los resultados es la disminución de tiempo de entrenamiento del clasificador, debido mayoritariamente a la disminución de características del descriptor con respecto a su implementación no uniforme. Al contrario que ocurriera para los otros dos clasificadores, la sensibilidad supera a la precisión. En términos absolutos, la exactitud del clasificador con este descriptor es inferior al obtenido con LBP, si bien en una cuantía casi imperceptible, aunque sigue superando de forma significativa al obtenido con HOG.

3.1.4 LBP-Uniforme + SVM: Optimización de parámetros

En este epígrafe buscaremos aquellos parámetros que mejores resultados dan para el SVM entrenado con el descriptor LBP-Uniforme de las imágenes. Así se efectúa una comparativa entre distintos clasificadores, cada uno de ellos con distinto núcleo y parámetros. La función que implementa dicha tarea se ha mostrado (**find_best_params**, Anexo: ***TestBasico.py***). Así, de la búsqueda de parámetros efectuada, se muestran aquellos con mejores resultados:

	mean_test_accuracy	mean_test_f1	mean_test_precision	mean_test_recall	\
4	0.989086	0.987810	0.986833	0.988824	
7	0.989086	0.987808	0.986828	0.988824	
6	0.986681	0.985114	0.983920	0.986340	
20	0.986312	0.984704	0.983102	0.986340	
19	0.986312	0.984704	0.983102	0.986340	
18	0.986312	0.984704	0.983102	0.986340	
15	0.986127	0.984484	0.983899	0.985098	
1	0.984647	0.982798	0.984238	0.981374	
3	0.983907	0.981958	0.983820	0.980131	
12	0.979282	0.976753	0.979627	0.973923	
10	0.970033	0.966416	0.967707	0.965230	
0	0.965594	0.961696	0.956993	0.966474	
13	0.958380	0.953066	0.961390	0.944950	
8	0.958010	0.954723	0.921892	0.990067	
5	0.958010	0.954723	0.921892	0.990067	

	param_C	param_gamma	param_kernel
4	10	0.01	rbf
7	100	0.01	rbf

6	100	0.001	rbf
20	100	NaN	linear
19	10	NaN	linear
18	1	NaN	linear
15	100	0.001	sigmoid
1	1	0.01	rbf
3	10	0.001	rbf
12	10	0.001	sigmoid
10	1	0.01	sigmoid
0	1	0.001	rbf
13	10	0.01	sigmoid
8	100	0.1	rbf
5	10	0.1	rbf

Al igual que ocurriese con los clasificadores entrenados con HOG y LBP, en este caso resulta también que el mejor kernel es en términos generales el Gaussiano. Los mejores resultados se encuentran algo por debajo al 99%, un resultado superior al obtenido con HOG pero algo pero al del LBP. Al igual que con el LBP, el kernel Lineal sigue siendo una buena alternativa en términos de rendimiento y bondad, pues se sitúa por delante del Sigmoidal.

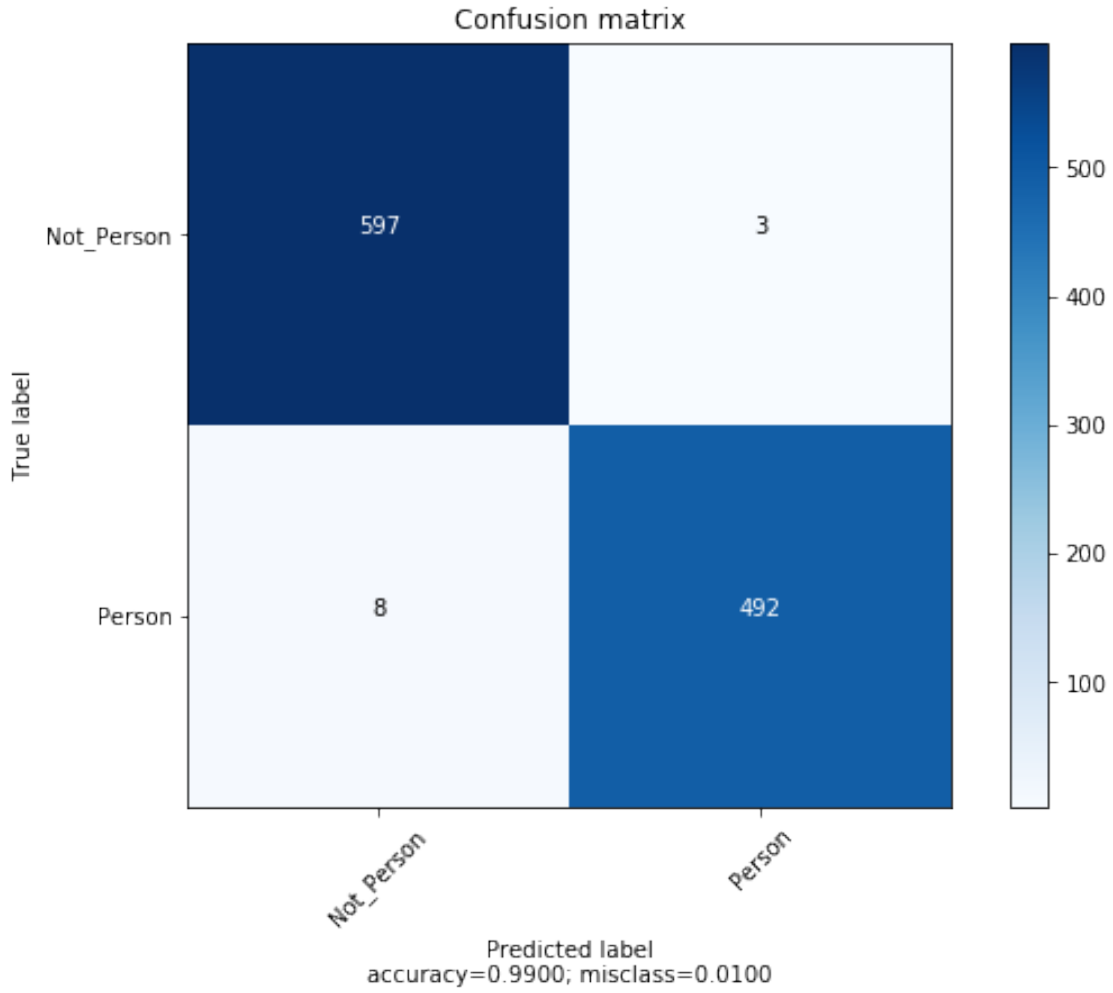
3.2 Combinación de características

En este epígrafe de la memoria, se lleva a cabo el análisis de los resultados obtenidos en los clasificadores SVM entrenados por medio de descriptores HOG y LBP de las imágenes de nuestro conjunto de datos. La implementación LBP utilizada es la creada en este mismo trabajo y el descriptor HOG es el propio de *OpenCV*.

3.2.1 HOG + LBP + SVM: Medidas de bondad del clasificador generado

En primer lugar, y al igual que se ha realizado para el resto de descriptores, se muestra el resultado del clasificador SVM con parámetros estándar (C = 1) y núcleo lineal:

Exactitud	F1-Score	Precision	Sensibilidad
0.99	0.988945	0.993939	0.984



A simple vista puede apreciarse como los resultados mejoran a los obtenidos con los otros clasificadores entrenados con HOG, LBP y U-LBP. En términos generales la clasificación es mejor, pues la exactitud llega al 99%, aunque tiene como aspecto negativo el ya sabido en cuanto que la clasificación de personas es algo peor que la de fondos (sensibilidad algo inferior a la precisión).

3.2.2 HOG + LBP + SVM: Validación cruzada

Para poder afianzar la realidad de los resultados obtenidos en el punto anterior, se lleva a cabo una validación cruzada con 10-fold, cuyos resultados para cada uno de los splits del conjunto de datos:

	fit_time	score_time	test_accuracy	test_f1	test_precision	test_recall
0	250.245140	74.394269	0.996303	0.995885	0.991803	1.000000
1	249.548985	75.540472	0.990758	0.989691	0.987654	0.991736
2	244.330669	74.816974	0.994455	0.993840	0.987755	1.000000
3	244.956663	73.578708	0.987061	0.985567	0.983539	0.987603
4	268.435083	86.246328	0.994455	0.993789	0.995851	0.991736

```
5  275.969329   83.923187       0.996303  0.995868        0.995868     0.995868
6  265.534541   84.658310       0.990741  0.989518        1.000000     0.979253
7  265.907844   86.506730       0.987037  0.985447        0.987500     0.983402
8  195.834891   64.559695       0.987037  0.985386        0.991597     0.979253
9  194.283220   64.930462       0.998148  0.997921        1.000000     0.995851
```

Del mismo modo, visualizamos los resultados medios para todos los splits:

```
     fit_time  score_time  test_accuracy   test_f1  test_precision  test_recall
9  245.504636   76.915514        0.99223  0.991291        0.992157      0.99047
```

Vemos como los resultados obtenidos de media en la validación cruzada son los mejores hasta el momento. La sensibilidad, que era nuestra única pega en el punto anterior vemos como es realmente mayor a la inicialmente vista. En general, para el núcleo Lineal y parámetros por defecto (C = 1), es una gran aproximación y se obtienen resultados al nivel de los mejores obtenidos en el resto de descriptores.

3.2.3 HOG + LBP + SVM: Optimización de parámetros

Al igual que con el resto de descriptores, efectuamos una búsqueda de los parámetros que maximicen las medidas de bondad para el clasificador SVM entrenado con LBP y HOG. Los resultados de dicha búsqueda son los mostrados a continuacíón:

```
    mean_test_accuracy  mean_test_f1  mean_test_precision  mean_test_recall  \
4             0.993711      0.992971             0.992167          0.993791
7             0.993711      0.992971             0.992167          0.993791
3             0.992971      0.992134             0.992145          0.992135
6             0.992971      0.992136             0.992151          0.992135
20            0.992416      0.991513             0.991730          0.991307
19            0.992416      0.991513             0.991730          0.991307
18            0.992416      0.991513             0.991730          0.991307
15            0.992231      0.991304             0.991727          0.990893
12            0.992231      0.991302             0.991725          0.990893
1             0.991676      0.990707             0.989317          0.992135
0             0.983537      0.981671             0.977509          0.985927
9             0.977802      0.975391             0.967098          0.983858
10            0.902886      0.893814             0.874824          0.913908
13            0.869404      0.854915             0.849718          0.860514
16            0.863855      0.848056             0.846707          0.849754

   param_C param_gamma param_kernel
4       10        0.01          rbf
7      100        0.01          rbf
3       10       0.001          rbf
6      100       0.001          rbf
20     100         NaN       linear
19      10         NaN       linear
```

```
18       1         NaN       linear
15     100       0.001      sigmoid
12      10       0.001      sigmoid
1        1        0.01          rbf
0        1       0.001          rbf
9        1       0.001      sigmoid
10       1        0.01      sigmoid
13      10        0.01      sigmoid
16     100        0.01      sigmoid
```

Al igual que para el resto de descriptores, el mejor clasificador SVM obtenido es aquel de núcleo Gaussiano con gamma de 0.01. Del mismo modo, el clasificador con núcleo Lineal sigue siendo una gran aproximación, pues en general se sitúa por delante del núcleo Sigmoidal y a muy poco del Gaussiano. El resultado del mejor clasificador de nuestra búsqueda (Núcleo Gaussiano, C = 10 y gamma = 0,01) se sitúa por delante del mejor obtenido anteriormente, por lo que podemos afirmar que la combinación de LBP y HOG es la que mejores resultados nos ha dado frente a los clasificadores entrenados con HOG, LBP y LBP-Uniforme de forma independiente.

3.3 Detección múltiple de peatones

En este último apartado del trabajo, se lleva a cabo el ejercicio propuesto de detectar varias personas en imágenes, independientemente del tamaño de la imagen o la escala de los peatones en la imagen. La aproximación finalmente implementada se ha compuesto de un análisis de diferentes ventanas sobre las distintas escalas de una misma imagen. Cada una de esas ventanas ha sido analizada en un clasificador SVM, previa extracción del descriptor, a fin de averiguar si ella se corresponde a la imagen de un peatón o de fondo.

El código de la función principal que se encarga de aplicar la detección múltiple de personas sobre un directorio de imágenes es el mostrado a continuación (Anexo: **TestBasico.py**):

```
def multi_target_person_detector(clf, descriptor):
    """Detect multiple person in images contained in data/person_detection.

    Args:
        (SVM) clf: Classifier to use for detection.
        (Descriptor) descriptor: descriptor to use for image feature extraction.
    """
    for file in os.listdir(PATH_MULTIPLE_PERSON):
        if file.startswith('.'): continue
        print file
        img = cv2.imread(PATH_MULTIPLE_PERSON + file, cv2.IMREAD_COLOR)
        person_detector(clf, img, descriptor, file)
    cv2.waitKey(0)
```

Tal y como se puede observar, en esta función se aplica la detección múltiple de peatones a un conjunto de imágenes contenida en un directorio concreto. Esta detección se llevará a cabo por medio del clasificador entrenado con el descriptor que se pase por segundo parámetro, a fin de que se pueda extraer ese mismo descriptor sobre cada una de las imágenes correspondientes a las

ventanas deslizantes, que se pasarán a su vez al clasificador para que determine si corresponde a un peatón o no.

Las funciones concretas que implementan la **detección de peatones en una imagen** se muestran a continuación (Anexo: **TestBasico.py**), siendo el más importante **person_detector**, pues define el algoritmo a seguir:

```
def person_detector(clf, img, descriptor, file):
    """Detect multiple person in an image.

    Args:
        (SVM) clf: Classifier to use for detection.
        (numeric[][]) img: Image to be used for detection.
        (Descriptor) descriptor: descriptor to use for image feature extraction.
        (String) file: Name of the file in which to find multiple person.
    """
    (win_w, win_h) = (64, 128)
    coors = []
    probs = []

    for resized in get_resizes(imutils.resize(img, int(img.shape[1] * 2)),
                                                                    scale=1.5):
        rt = img.shape[1] / float(resized.shape[1])
        (win_w_r, win_h_r) = (win_w * rt, win_h * rt)

        for (x, y, window) in get_window_coor(resized, step=32, w_size=(win_w,
                                                                    win_h)):
            if window.shape[0] != win_h or window.shape[1] != win_w: continue
            img_d = descriptor.compute(window)
            data = [img_d.flatten()]

            prob = clf.predict_proba(data)[0, 1]

            if prob > 0.7:
                coor = [int(x * rt), int(y * rt), int(x * rt + win_w_r),
                    int(y * rt + win_h_r)]
                coors.append(coor)
                probs.append(prob)

    boxes = non_max_suppression_fast(np.array(coors), 0.3)

    for x_s, y_s, x_e, y_e in boxes:
        cv2.rectangle(img, (x_s, y_s), (x_e, y_e), (0, 255, 0), 2)

    cv2.namedWindow("Person detection_" + file, cv2.WINDOW_AUTOSIZE)
    cv2.imshow("Person detection_" + file, img)
    cv2.waitKey(1)

def get_window_coor(image, step, w_size):
```

```
    """Get sliding window coordinates for an given image,
    window and step size.

    Args:
        (numeric[][]) image: Image where to compute sliding windows
        coordinates.
        (int) step: Pixel difference between windows.
        (int[]) w_size: Size of the required windows.
    Returns:
        (int[]): Coordinates for each window.
    """
    coor = list(product(*[range(0, image.shape[0], step),
        range(0, image.shape[1], step)]))
    for y, x in coor: yield (x, y, image[y:y + w_size[1], x:x + w_size[0]])


def get_resizes(image, scale=1.5, min_size=(64, 128)):
    """Get different image resizes of an original  image.

    Args:
        (numeric[][]) image: Image to resize.
        (float) scale: factor of resize.
        (int[]) min_size: Min size of the resized image.
    Returns:
        (numeric[][]): Resized images.
    """
    while True:
        yield image
        if (image.shape[0] < min_size[1]) or (image.shape[1] < min_size[0]): break
        else: image = imutils.resize(image, int(image.shape[1] / scale


def non_max_suppression_fast(boxes, overlap_thresh):
    """Extracted from Malisiewicz et al.
    (https://github.com/quantombone/exemplarsvm)

    Remove overlapped bounding boxes in an image.

    Args:
        (int[][]) boxes: Bounding boxes coordinates.
        (float) overlap_thresh: Thresh to remove two overlapped images.
    """
    # if there are no boxes, return an empty list
    if len(boxes) == 0:
        return []

    # if the bounding boxes integers, convert them to floats --
    # this is important since we'll be doing a bunch of divisions
```

```
if boxes.dtype.kind == "i":
    boxes = boxes.astype("float")

# initialize the list of picked indexes
pick = []

# grab the coordinates of the bounding boxes
x1 = boxes[:, 0]
y1 = boxes[:, 1]
x2 = boxes[:, 2]
y2 = boxes[:, 3]

# compute the area of the bounding boxes and sort the bounding
# boxes by the bottom-right y-coordinate of the bounding box
area = (x2 - x1 + 1) * (y2 - y1 + 1)
idxs = np.argsort(y2)

# keep looping while some indexes still remain in the indexes
# list
while len(idxs) > 0:
    # grab the last index in the indexes list and add the
    # index value to the list of picked indexes
    last = len(idxs) - 1
    i = idxs[last]
    pick.append(i)

    # find the largest (x, y) coordinates for the start of
    # the bounding box and the smallest (x, y) coordinates
    # for the end of the bounding box
    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]])

    # compute the width and height of the bounding box
    w = np.maximum(0, xx2 - xx1 + 1)
    h = np.maximum(0, yy2 - yy1 + 1)

    # compute the ratio of overlap
    overlap = (w * h) / area[idxs[:last]]

    # delete all indexes from the index list that have
    idxs = np.delete(idxs, np.concatenate(([last],
                                           np.where(overlap > overlap_thresh)[0])))

# return only the bounding boxes that were picked using the
# integer data type
return boxes[pick].astype("int")
```

De forma resumida, el algoritmo seguido ha sido el siguiente:

1. primer lugar, se obtiene un redimensionado de la imagen original al 200% (Ampliamos para detectar personas pequeñas en relación a la imagen), que le es pasado al método **get_resizes**. Este método devuelve todos los tamaños posibles para esa imagen obtenidos por medio de la aplicación de un factor de reducción.

2. Seguidamente, sobre un tamaño de la misma imagen dado por la función anterior, obtenemos las posibles coordenadas de la ventana deslizante. Para ello, hacemos uso de la función **get_window_coor**, al cual le decimos la diferencian en píxeles entre una ventana y otra (paso que debe aplicar).

3. Para cada una de las ventanas obtenidas sobre un dimensionado concreto de la imagen original y según las coordenadas posibles calculadas, se calcula su descriptor, que será sobre aquel pasado por parámetro y posteriormente se llevará a cabo su clasificación. Este clasificador es igualmente pasado por argumento y deberá estar entrenado sobre datos con mismo descriptor al anterior.

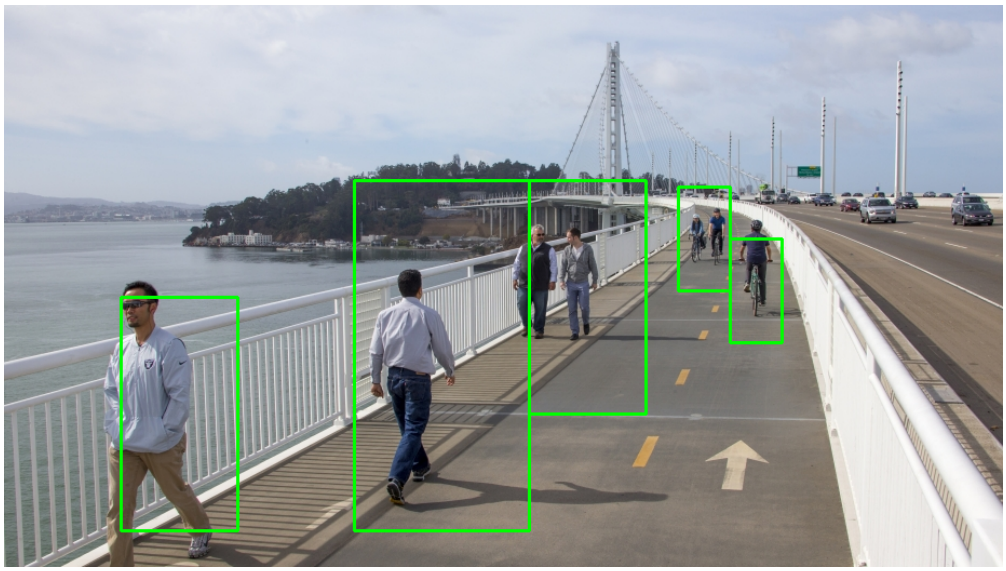
4. A continuación, sobre cada una de las ventanas cuyo descriptor una vez clasificado apunta a que se corresponde a una persona, se lleva a cabo un proceso denominado *non-maximun supression*, implementado en la función con mismo nombre. Esta función selecciona de aquellas ventanas que se refieren a una misma detección, aquellas de mayor importancia de acuerdo al solapamiento de estas entre sí.

5. Por último, se pinta un rectángulo que trace el borde de las ventanas seleccionadas como de mayor importancia a raíz de la función anterior.

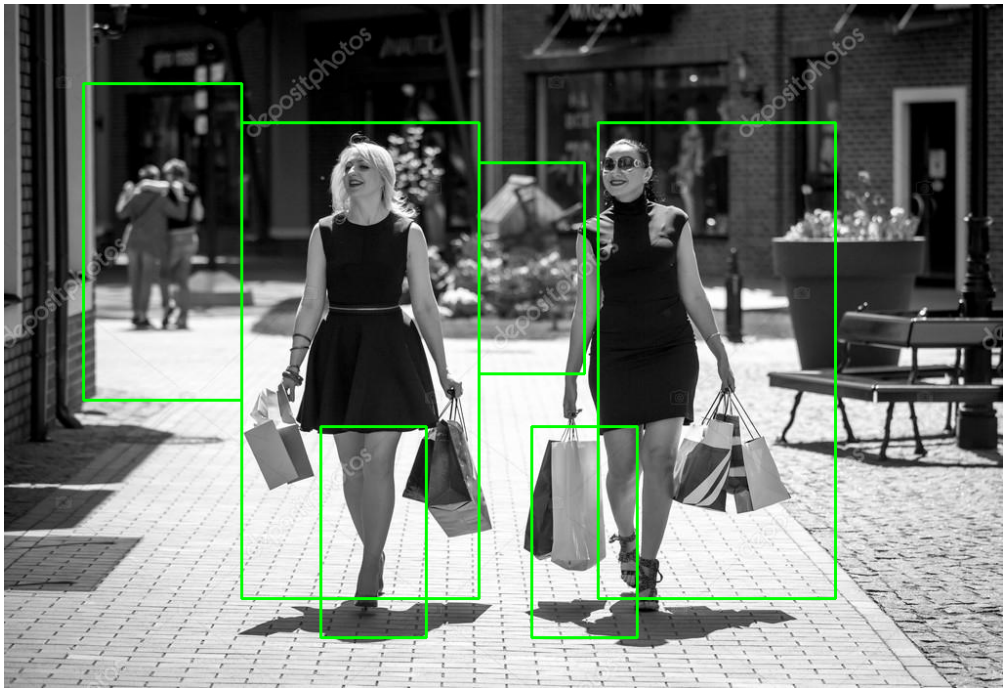
3.3.1 Resultados

A continuación, se presentan varios ejemplos resultantes de la ejecución de la función anterior para la detección de mútiple peatones:

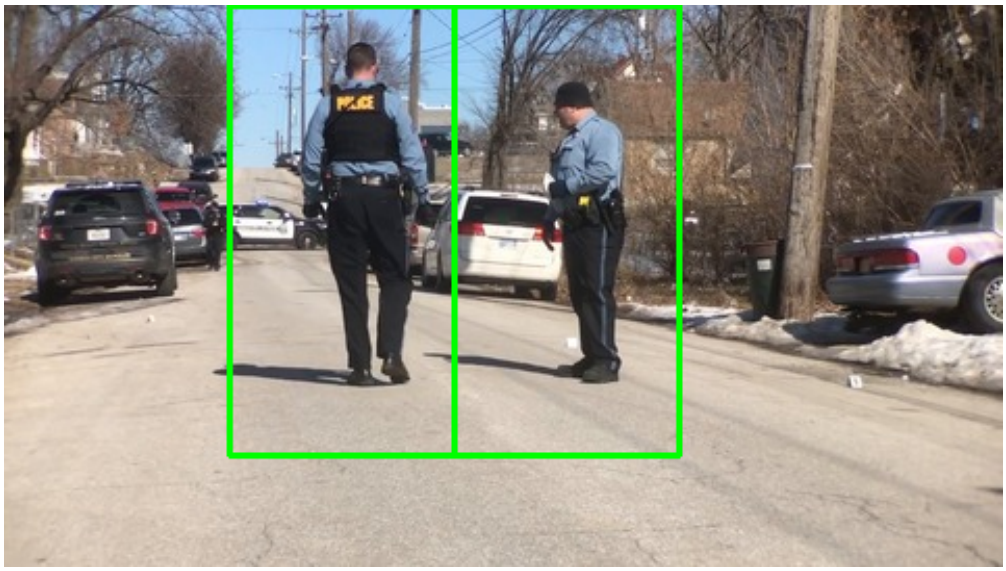
Ejemplo 1:



Ejemplo 2:



Ejemplo 3:



Interpretación:

Tal y como se puede observar, la detección de personas no es mala del todo, si bien es cierto que se generan ciertos Falsos Positivos (Ejemplo 2) y los rectángulos resultantes no son del todo definitorios de la detección realizada. Este segundo problema ha sido quizás el más crítico, pues el primero depende de la efectividad del clasificador y parece que también del conjunto de datos con el que ha sido entrenado, algo que está fuera de nuestra mano. Sin embargo y si bien en la bibliografía no se han encontrado algoritmos de selección de bounding boxes que den un resultado más que el decente arriba mostrado, creo que sería posible como una mejora a la aproximación aquí empleada proponer otro más efectivo en caso de que se lleve a cabo un proyecto de similares características en un futuro. Por este hecho y por la debilidad del algoritmo de selección de cajas, podemos observar como en el ejemplo 1 se detectan en un mismo rectángulo a dos peatones, sucediendo esto en dos ocasiones distintas. En el ejemplo 2, por su parte se detectan como personas las piernas de un peatón y la bolsa que porta otro de los peatones.

4 Anexo

A continuación, se muestra todo el código generado en la realización de este proyecto, que a su vez compone la aplicación final implementada.

4.1 LBPDescriptor.py

```
import cv2 as cv2
import _functions

class LBPDescriptor:

    #Local Binary Pattern descriptor object class

    def compute(self, img):
        """Compute descriptor.

        Args:
            img (int[][][]): An RGB image.
        Returns:
            float[]: Local Binary Pattern descriptor.
        """
        grey_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        feat = _functions.lbp(grey_img)
        return feat
```

4.2 UniformLBPDescriptor.py

```
import cv2 as cv2
import _functions
import time
import numpy as np

class UniformLBPDescriptor:

    """Uniform Local Binary Pattern descriptor object class

    Attributes:
        (int[]) uniform_patterns: LBP histogram uniform values.
    """

    uniform_patterns = []

    def __init__(self):
        """Initialize descriptor.

        Initialize descriptor and compute uniform patterns.
        """
        self.uniform_patterns = self.__get_uniform_patterns()

    def compute(self, img):
        """Compute descriptor.

        Args:
            (int[][][]) img: An RGB image.
        Returns:
            float[]: Uniform Local Binary Pattern descriptor.
        """
        grey_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        feat = _functions.uniform_lbp(grey_img, self.uniform_patterns)
        return feat

    def __get_uniform_patterns(self):
        """Compute uniform patterns.

        Compute uniform patterns (Number of transitions        <= 2) and update instance

        45 ->  00101101:
            -Number of transitions: 6 (Non-uniform)

        255 -> 11111110:
            -Number of transitions: 2 (Uniform)

        Returns:
```

```
    int[]: Uniform patterns for descriptor instance.
"""
patterns = np.zeros(256)
count = 0
for i in range(2**8):
    binary = map(int, format(i, '08b'))
    first = prev = tran = 0
    for j in range(binary.__len__()):
        if j == 0:
            first = binary[j]
            prev = binary[j]
        tran = tran + 1 if
            binary[j] != prev else tran
        tran = tran + 1 if (j == (binary.__len__() - 1)) & (binary[j] != first)
        prev = binary[j]
    if tran <= 2:
        count += 1
        patterns[i] = count
return patterns
```

4.3 _functions.pyx

```
import numpy as np
import cv2 as cv2


cpdef lbp(unsigned char[:,:] img):
    """Local Binary Pattern calculation.

    Args:
        (unsigned char[:,:]) img: Greyscale image.
    Returns:
        (double[:]) histograms: Histograms of image blocks.
    """
    cdef int y_lim = img.shape[0]
    cdef int x_lim = img.shape[1]
    cdef double[:,:] texture_map = get_texture_map(img)

    hists = []

    for y in range(0, y_lim - 8, 8):
        for x in range(0, x_lim - 8, 8):
            hist = compute_block_lbp(texture_map, y, x)
            hists.append(hist/np.linalg.norm(hist))

    return np.concatenate(hists)


cpdef uniform_lbp(unsigned char[:,:] img, double[:] uniform_patterns):
    """Uniform Local Binary Pattern calculation.

    Args:
        (unsigned char[:,:]) img: Greyscale image.
        (list) uniform_patterns: List of considered uniform patterns.
    Returns:
        (double[:]) histograms: Histograms of image blocks.
    """
    cdef int y_lim = img.shape[0]
    cdef int x_lim = img.shape[1]
    cdef double[:,:] texture_map = get_texture_map(img)

    hists = []

    for y in range(0, y_lim - 8, 8):
        for x in range(0, x_lim - 8, 8):
            hist = compute_block_ulbp(texture_map, y, x, uniform_patterns)
            hists.append(hist/np.linalg.norm(hist))
```

```
    return np.concatenate(hists)


cpdef get_texture_map(unsigned char[:,:] img):
    """LBP texture map calculation.

    Args:
        (unsigned char[:,:]) img: Greyscale image.
    Returns:
        (double[:,:]) texture_map: Texture map with LBP.
    """
    cdef int y_lim = img.shape[0]
    cdef int x_lim = img.shape[1]
    cdef double[:,:] texture_map = np.zeros((y_lim, x_lim))
    cdef int val = 0
    cdef int pt = 0

    for y in range(1, y_lim - 1):
        for x in range(1, x_lim - 1):
            val = img[y, x]
            pt = 0
            pt = pt | (1 << 7) if val <= img[y - 1, x - 1] else pt
            pt = pt | (1 << 6) if val <= img[y - 1, x] else pt
            pt = pt | (1 << 5) if val <= img[y - 1, x + 1] else pt
            pt = pt | (1 << 4) if val <= img[y, x + 1] else pt
            pt = pt | (1 << 3) if val <= img[y + 1, x + 1] else pt
            pt = pt | (1 << 2) if val <= img[y + 1, x] else pt
            pt = pt | (1 << 1) if val <= img[y + 1, x - 1] else pt
            pt = pt | (1 << 0) if val <= img[y, x - 1] else pt
            texture_map[y, x] = pt

    texture_map[0,:] = texture_map[1, :]
    texture_map[texture_map.shape[0]-1,:] = texture_map[texture_map.shape[0]-2,:]
    texture_map[:,0] = texture_map[:, 1]
    texture_map[:,texture_map.shape[1]-1] = texture_map[:,texture_map.shape[1]-2]

    return texture_map


cpdef compute_block_lbp(double[:,:] texture_map, int i_start, int j_start):
    """Local Binary Pattern image block calculation.

    Args:
        (int) i_start: Block x start index.
        (int) j_start: Block y start index.
        (double[:,:]) texture_map: Texture map with LBP.
    Returns:
        (double[:]) histogram: Histogram for each image block.
```

```
    """
    cdef double[:] hist = np.zeros(256)
    cdef double val = 0
    cdef int i
    cdef int j

    for i in range(i_start, i_start + 16) :
        for j in range(j_start, j_start + 16) :
            val = texture_map[i,j]
            hist[(<int>val)] = hist[(<int>val)] + 1

    return hist


cpdef compute_block_ulbp(double[:,:] texture_map, int i_start,
                         int j_start, double[:] uniform_patterns):
    """Uniform Local Binary Pattern image block calculation.

    Args:
        (int) i_start: Block x start index.
        (int) j_start: Block y start index.
        (double[:,:]) texture_map: Texture map with LBP.
    Returns:
        (double[:]) histogram: Histogram for each image block.
    """

    cdef double[:] hist = np.zeros(59)
    cdef int val = 0
    cdef double index = 0.0
    cdef int i
    cdef int j

    for i in range(i_start, i_start + 16) :
        for j in range(j_start, j_start + 16) :
            val = (<int>texture_map[i, j])
            if uniform_patterns[val] > 0:
                index = uniform_patterns[val]-1
                hist[(<int>index)] = hist[(<int>index)] + 1.0
            else:
                hist[58] = hist[58] + 1.0
    return hist
```

4.4 TestBasico.py

```
# coding=utf-8

import cv2 as cv2
import numpy as np
import os
import LBPDescriptor as LBP
import UniformLBPDescriptor as ULBP
from sklearn import metrics, svm
from sklearn.model_selection import cross_validate, GridSearchCV
import cPickle
import imutils
from itertools import product
import pandas as pd
import warnings
import matplotlib.pyplot as plt

PATH_POSITIVE_TRAIN = "../data/train/pedestrians/"
PATH_NEGATIVE_TRAIN = "../data/train/background/"
PATH_POSITIVE_TEST = "../data/test/pedestrians/"
PATH_NEGATIVE_TEST = "../data/test/background/"
PATH_MULTIPLE_PERSON = "../data/person_detection/"
EXAMPLE_POSITIVE = PATH_POSITIVE_TEST + "AnnotationsPos_0.000000_crop_000011b_0.png"
EXAMPLE_NEGATIVE = PATH_NEGATIVE_TEST + "AnnotationsNeg_0.000000_00000002a_0.png"


def __main__():
    ### Histogram of Gradients (HoG)
    process(['hog'])
    ### Local binary pattern (LBP)
    process(['lbp'])

    ### Uniform Local Binary Pattern (ULBP)
    process(['ulbp'])
    ### Local Binary Pattern + Histogram of Gradients (LBP + HoG)
    process(['lbp', 'hog'])

    ## Multiple person detection
    clf = get_custom_SVM(['ulbp'], kernel='rbf', gamma=0.01, C=10)
    multi_target_person_detector(clf, ULBP.UniformLBPDescriptor())


def get_custom_SVM(descriptors, gamma=None, C=1, kernel='linear'):
    """Get SVM model for given parameters and descriptor(s)

    Args:
        (String[]) descriptors: Descriptors
```

```
        (float) gamma: SVM gamma parameter (Default = None)
        (float) C: SVM C parameter (Default = 1)
        (String) kernel: SVM kernel (Default = 'linear')
    """
    data, classes = load_data(descriptors, orig_train_test=False)
    idx = np.random.permutation(len(data))
    x, y = data[idx], classes[idx]
    clf = svm.SVC(kernel=kernel, gamma=gamma, C=C, probability=True)
    clf.fit(x, y)
    print "--> Clasificador entrenado"
    return clf


def process(descriptors):
    """Execute process for:

    1.  Calculate standard SVM and performance metrics
    2.  Calculate standard SVM and 10-fold CV performance metrics
    3.  Parameter search grid for SVM and performance metrics

    Args:
        (String[]) descriptors: Image descriptors for making process.
    """

    print "----> Loading data for " + ' '.join(descriptors) + "descriptors"
    data_train, data_test, classes_train, classes_test = load_data(descriptors,
        orig_train_test=True)
    data, classes = load_data(descriptors, orig_train_test=False)

    print "----> SVM con parámetros estándar (" + ' '.join(descriptors) + "):"
    print standard_svm(data_train, data_test, classes_train, classes_test, save=True,
        name="svm_std_" + '_'.join(descriptors))

    print "----> SVM con 10-fold CV y parámetros estándar (" +
        ' '.join(descriptors) + "):"
    print cv_standard_svm(data, classes, save=True, name="svm_10cv_std_" +
        '_'.join(descriptors))

    print "----> Búsqueda mejores parámetros SVM con 5-fold CV (" +
        ' '.join(descriptors) + "):"
    print find_best_params(data, classes, save=True, name="svm_5cv_grid_" +
        '_'.join(descriptors))


def standard_svm(data_train, data_test, classes_train, classes_test, save=False,
                 name=None):
    """Compute standard linear SVM classifier and metrics for a given
    train-test data set.
```

```
    Compute Accuracy, precision, recall, F1 and print confusion
    matrix for Linear standard SVM and given train-test data set.

    Args:
        (float[][]) data_train: Image descriptors of train data.
        (float[][]) data_test: Image descriptors of test data.
        (int[]) classes_train: Real label of the train data.
        (int[]) classes_test: Real label of the test data.
        (bool) save: Save CV score.
        (String) name: name of score file.
    """
    clf = train(data_train, classes_train)
    prediction = test(data_test, clf)
    std_clf_metrics(classes_test, prediction, save=save, name=name)


def std_clf_metrics(classes_test, prediction, save=False, name=None):
    """Compute metrics for a given prediction.

    Compute Accuracy, precision, recall, F1 and print confusion
    matrix for given prediction.

    Args:
        (int[]) classes_test: Real label of the data.
        (int[]) prediction: Predicted label of the data.
        (bool) save: Save CV score.
        (String) name: name of score file.
    """
    scores = {"Exactitud": [metrics.accuracy_score(classes_test, prediction)],
              "Precision": [metrics.precision_score(classes_test, prediction)],
              "Sensibilidad": [metrics.recall_score(classes_test, prediction)],
              "F1-Score": [metrics.f1_score(classes_test, prediction)]}
    df = pd.DataFrame.from_dict(scores)
    print(df)
    cm = (metrics.confusion_matrix(classes_test, prediction))
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        plot_confusion_matrix(cm, normalize=False)
    if save:
        write_data(df, "../scores/" + name + ".pkl")
        write_data(cm, "../scores/" + name + "_cm.pkl")


def cv_standard_svm(data, classes, save=True, name=None):
    """Compute CV with default parameters.

    Compute cross-validation for standard linear SVM classifier.
```

```
    Args:
        (float[][]) data: Image descriptors.
        (int[]) classes: Label of the data.
        (bool) save: Save CV score.
        (String) name: name of score file.
        (int) cv: Number of folds.
    Returns:
        (DataFrame): Scores for each CV split.
    """
    clf = svm.SVC(kernel='linear')
    scoring = ['accuracy', 'precision', 'recall', 'f1']
    scores = cross_validate(clf, data, classes, cv=10, n_jobs=-1,
                            scoring=scoring, verbose=10, return_train_score=False)
    df = pd.DataFrame.from_dict(scores)
    if save: write_data(df, '../scores/' + name + ".pkl")
    return df


def find_best_params(data, classes, save=True, name=None):
    """SVM parameter tuning.

    Search best kernel and parameters for SVM classifier.

    Args:
        (float[][]) data: Image descriptors.
        (int[]) classes: Label of the data.
        (bool) save: Save CV score.
        (String) name: name of score file.
    Returns:
        (DataFrame): Scores for kernel & parameter combination.
    """
    tuned_parameters = [{'kernel': ['rbf'], 'gamma': [0.001, 0.01, 0.1],
                         'C': [1, 10, 100]},
                        {'kernel': ['sigmoid'], 'gamma': [0.001, 0.01, 0.1],
                         'C': [1, 10, 100]},
                        {'kernel': ['linear'], 'C': [1, 10, 100]}]
    svc = svm.SVC()
    scoring = ['accuracy', 'precision', 'recall', 'f1']
    clf = GridSearchCV(svc, tuned_parameters, cv=5, verbose=10, n_jobs=-1,
                       return_train_score=False, scoring=scoring,
                       refit='accuracy')
    res = clf.fit(data, classes)
    df = pd.DataFrame.from_dict(res.cv_results_).iloc[:, 2:13]
    print df
    if save: write_data(df, '../scores/' + name + ".pkl")
```

```
def train(data_train, classes):
    """Compute standard linear SVM classifier.

    Args:
        (float[][]) data_train: Image descriptors of train data.
        (int[]) classes: Label of the train data.
    Returns:
        (SVM): Trained SVM classifier.
    """
    idx = np.random.permutation(len(data_train))
    x, y = data_train[idx], classes[idx]
    clf = svm.SVC(kernel='linear')
    clf.fit(x, y)
    return clf


def test(data_test, classifier):
    """Compute predictions for a given classifier.

    Args:
        (float[][]) data_test: Image descriptors of test data.
        (SVM) classifier: SVM classifier to use.
    Returns:
        (int[]): Predictions for given data and classifier.
    """
    prediction = classifier.predict(data_test)
    return prediction


def load_data(descriptor_names, orig_train_test=False):
    """Load image descriptors data set.

    Args:
        (String[]) descriptor_names: Name of the descriptors
        to use for loading each image.
        (bool) orig_train_test: Keep original train/test split.
    Returns:
        (float[][]) data_train: Image descriptors of train data.
        (float[][]) data_test: Image descriptors of test data.
        (int[]) classes_train: Labels for train images.
        (int[]) classes_test: Labels for test images.
    """
    switcher = {
        "hog": cv2.HOGDescriptor(),
        "lbp": LBP.LBPDescriptor(),
        "ulbp": ULBP.UniformLBPDescriptor()
    }
    descriptors = [switcher.get(descriptor_name) for descriptor_name
```

```
            in descriptor_names]
    data_train, classes_train = load_with_descriptor(PATH_POSITIVE_TRAIN,
            1, descriptors)
    data_aux, classes_aux = load_with_descriptor(PATH_NEGATIVE_TRAIN, 0, descriptors)
    data_train = np.concatenate((data_train, data_aux), axis=0)
    classes_train = np.append(classes_train, classes_aux)
    data_test, classes_test = load_with_descriptor(PATH_POSITIVE_TEST, 1, descriptors)
    data_aux, classes_aux = load_with_descriptor(PATH_NEGATIVE_TEST, 0, descriptors)
    data_test = np.concatenate((data_test, data_aux), axis=0)
    classes_test = np.append(classes_test, classes_aux)

    if orig_train_test:
        return data_train, data_test, classes_train, classes_test
    else:
        return np.concatenate((data_train, data_test), axis=0),
                np.append(classes_train, classes_test)


def load_with_descriptor(path, label, descriptors):
    """Load image descriptors data set.

    Args:
        (String) path: Path of the images folder.
        (int) label: Label of the images of the given path.
        (Descriptor[]) descriptors: Descriptors to use.
    Returns:
        (float[][]) data: Image descriptors of the data.
        (int[]) classes: Labels for images.
    """
    data = []
    classes = []
    lab = np.ones((1, 1), dtype=np.int32) if label == 1 else np.zeros((1, 1),
            dtype=np.int32)
    for file in os.listdir(path):
        if file.startswith('.'): continue
        img = cv2.imread(path + file, cv2.IMREAD_COLOR)
        img_d = [descriptor.compute(img).flatten() for descriptor in descriptors]
        data.append(np.concatenate(img_d))
        classes.append(lab)
        print file
    data = np.array(data)
    classes = np.array(classes, dtype=np.int32)
    return data, classes


def read_data(path):
    """Read .pkl file saved locally.
```

```
    Args:
        (String) path: Path of the file to load.
    Returns:
        (Object): Read file object.

    """
    with open(path, 'rb') as fid:
        return cPickle.load(fid)


def write_data(clf, path):
    """Save .pkl file locally.

    Args:
        (Object): Object to be saved.
    """
    with open(path, 'wb') as fid:
        cPickle.dump(clf, fid)


def plot_confusion_matrix(cm, target_names=['Not_Person', 'Person'],
                          title='Confusion matrix', normalize=True):
    """Plot confusion matrix with better aesthetic.

    Args:
        (int[][]) cm: Confusion matrix to plot.
        (String[]) target_names: Name of the matrix labels
        (String) title: Title to print above the matrix.
        (bool) normalize: Set if is matrix normalization required.
    """
    accuracy = np.trace(cm) / float(np.sum(cm))
    misclass = 1 - accuracy

    cmap = plt.get_cmap('Blues')

    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()

    if target_names is not None:
        tick_marks = np.arange(len(target_names))
        plt.xticks(tick_marks, target_names, rotation=45)
        plt.yticks(tick_marks, target_names)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
```

```
    thresh = cm.max() / 1.5 if normalize else cm.max() / 2
    for i, j in product(range(cm.shape[0]), range(cm.shape[1])):
        if normalize:
            plt.text(j, i, "{:0.4f}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")
        else:
            plt.text(j, i, "{:,}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label\naccuracy={:0.4f}; misclass={:0.4f}'.format(accuracy,
            misclass))
    plt.show()


def get_window_coor(image, step, w_size):
    """Get sliding window coordinates for an given image,
    window and step size.

    Args:
        (numeric[][]) image: Image where to compute sliding windows
        coordinates.
        (int) step: Pixel difference between windows.
        (int[]) w_size: Size of the required windows.
    Returns:
        (int[]): Coordinates for each window.
    """
    coor = list(product(*[range(0, image.shape[0], step),
            range(0, image.shape[1], step)]))
    for y, x in coor: yield (x, y, image[y:y + w_size[1], x:x + w_size[0]])


def get_resizes(image, scale=1.5, min_size=(64, 128)):
    """Get different image resizes of an original  image.

    Args:
        (numeric[][]) image: Image to resize.
        (float) scale: factor of resize.
        (int[]) min_size: Min size of the resized image.
    Returns:
        (numeric[][]): Resized images.
    """
    while True:
        yield image
        if (image.shape[0] < min_size[1]) or (image.shape[1] < min_size[0]): break
        else: image = imutils.resize(image, int(image.shape[1] / scale))
```

```
def multi_target_person_detector(clf, descriptor):
    """Detect multiple person in images contained in data/person_detection.

    Args:
        (SVM) clf: Classifier to use for detection.
        (Descriptor) descriptor: descriptor to use for image feature extraction.
    """
    for file in os.listdir(PATH_MULTIPLE_PERSON):
        if file.startswith('.'): continue
        print file
        img = cv2.imread(PATH_MULTIPLE_PERSON + file, cv2.IMREAD_COLOR)
        person_detector(clf, img, descriptor, file)
    cv2.waitKey(0)


def person_detector(clf, img, descriptor, file):
    """Detect multiple person in an image.

    Args:
        (SVM) clf: Classifier to use for detection.
        (numeric[][]) img: Image to be used for detection.
        (Descriptor) descriptor: descriptor to use for image feature extraction.
        (String) file: Name of the file in which to find multiple person.
    """
    (win_w, win_h) = (64, 128)
    coors = []
    probs = []

    for resized in get_resizes(imutils.resize(img, int(img.shape[1] * 2)), scale=1.5):
        rt = img.shape[1] / float(resized.shape[1])
        (win_w_r, win_h_r) = (win_w * rt, win_h * rt)

        for (x, y, window) in get_window_coor(resized, step=32, w_size=(win_w, win_h)):
            if window.shape[0] != win_h or window.shape[1] != win_w: continue
            img_d = descriptor.compute(window)
            data = [img_d.flatten()]

            prob = clf.predict_proba(data)[0, 1]

            if prob > 0.7:
                coor = [int(x * rt), int(y * rt), int(x * rt + win_w_r),
                        int(y * rt + win_h_r)]
                coors.append(coor)
                probs.append(prob)

    boxes = non_max_suppression_fast(np.array(coors), 0.3)
```

```
    for x_s, y_s, x_e, y_e in boxes:
        cv2.rectangle(img, (x_s, y_s), (x_e, y_e), (0, 255, 0), 2)

    cv2.namedWindow("Person detection_" + file, cv2.WINDOW_AUTOSIZE)
    cv2.imshow("Person detection_" + file, img)
    cv2.waitKey(1)


def non_max_suppression_fast(boxes, overlap_thresh):
    """Extracted from Malisiewicz et al.
    (https://github.com/quantombone/exemplarsvm)

    Remove overlapped bounding boxes in an image.

    Args:
        (int[][]) boxes: Bounding boxes coordinates.
        (float) overlap_thresh: Thresh to remove two overlapped images.
    """
    # if there are no boxes, return an empty list
    if len(boxes) == 0:
        return []

    # if the bounding boxes integers, convert them to floats --
    # this is important since we'll be doing a bunch of divisions
    if boxes.dtype.kind == "i":
        boxes = boxes.astype("float")

    # initialize the list of picked indexes
    pick = []

    # grab the coordinates of the bounding boxes
    x1 = boxes[:, 0]
    y1 = boxes[:, 1]
    x2 = boxes[:, 2]
    y2 = boxes[:, 3]

    # compute the area of the bounding boxes and sort the bounding
    # boxes by the bottom-right y-coordinate of the bounding box
    area = (x2 - x1 + 1) * (y2 - y1 + 1)
    idxs = np.argsort(y2)

    # keep looping while some indexes still remain in the indexes
    # list
    while len(idxs) > 0:
        # grab the last index in the indexes list and add the
        # index value to the list of picked indexes
        last = len(idxs) - 1
```

```
        i = idxs[last]
        pick.append(i)

        # find the largest (x, y) coordinates for the start of
        # the bounding box and the smallest (x, y) coordinates
        # for the end of the bounding box
        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]])

        # compute the width and height of the bounding box
        w = np.maximum(0, xx2 - xx1 + 1)
        h = np.maximum(0, yy2 - yy1 + 1)

        # compute the ratio of overlap
        overlap = (w * h) / area[idxs[:last]]

        # delete all indexes from the index list that have
        idxs = np.delete(idxs, np.concatenate(([last],
                                               np.where(overlap > overlap_thresh)[0])))

    # return only the bounding boxes that were picked using the
    # integer data type
    return boxes[pick].astype("int")
```