Extracción de Características en Imágenes 2024-2025

Master Ciencia de Datos

Universidad de Granada

Práctica 1

Miguel García López

Índice

Índice de figuras

Índice de cuadros

1. Camino

El camino escogido en las tareas a realizar es: $(1, 4, 6, 8)$.

2. Introducción

En la presente práctica de la asignatura de **Extracción de Características en Imágenes**, se llevarán a cabo una serie de tareas definidas por un grafo de decisión (fig 1). Dado este grafo es necesario seguir el camino hasta al final y allí donde haya una bifurcación, escoger entre una tarea básica o una tarea bonificadora. Las tareas bonificadores son iguales que las básicas, pero con un toque de dificultad y desarrollo por parte del alumno. De forma resumida, las tareas a realizar son las siguientes:

- **Búsqueda de un conjunto de datos:** El estudiante puede usar el *dataset* **MNIST** por defecto, pero en este caso se ha optado por la tarea complementaria de escoger uno.
- **Clasificación con HOG:** Se entrenará un modelo **SVM** usando el descriptor **HOG** y se realizará un análisis de los resultados del mismo. Además se aplicarán técnicas como validación cruzada y selección de hiperparámetros.
- **Clasificación con LBP:** Se realizará lo mismo que con **HOG** descrito en el apartado anterior. Además, como parte de la tarea complementaria bonificada, se usará una implementación propia de **LBP** para extraer las características del *dataset*.

De las 24000 imágenes se han escogido de forma aleatoria, y teniendo en cuenta equilibrio entre clases, 8000 imágenes para la clasificación con **SVM+LBP** y **SVM+HOG**.

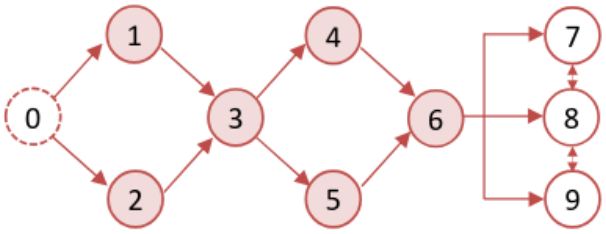

Figura 1: Grafo de decisión de tareas.

3. Dataset

Para el *dataset* se ha escogido el conjunto de "Gatos vs Perros" de *Kaggle*. Este se puede encontrar en el siguiente enlace: *https://www.kaggle.com/competitions/dogs-vs-cats*.

El conjunto contiene cerca de 24000 imágenes, la mitad de perros y la mitad de gatos. Este conjunto se compone de imágenes de multitud de resoluciones, por lo que se ha procedido a realizar varias pruebas en el código y se ha llegado a la conclusión de que 28×28 píxeles es un tamaño con el que poder trabajar por los siguiente motivos:

- La extracción de descriptores en imágenes de alta resolución lleva a altos tiempos de cómputo.
- La implementación de **LBP** es rápida, pero no tanto como las implementaciones de otros descriptores como **HOG** en *OpenCV*.
- Se han realizado varias pruebas y con tamaños de resolución mucho mayores no se consiguen unos resultados mucho mejores (hasta donde se ha podido comprobar).

Además de lo descrito, se han transformado las imágenes a escala de grises para trabajar con un solo canal.

4. Clasificación con HOG

4.1. HOG

El descriptor **HOG** es una técnica ampliamente utilizada en la visión por computador, cuyo objetivo es capturar la estructura local de las imágenes basándose en los gradientes de intensidad. Divide la imagen (fig 2) en celdas pequeñas y calcula un histograma de orientaciones de gradiente dentro de cada celda. Para mejorar la robustez frente a cambios de iluminación, se normalizan los histogramas en bloques de celdas adyacentes.

4.2. Búsqueda de hiperparámetros

Se ha implementado una búsqueda de hiperparámetros de forma que es posible buscar solo hiperparámetros del algoritmos **SVM** o, si se selecciona, búsqueda para **SVM** y **HOG**. Hay que tener en cuenta que la búsqueda de hiperparámetros es un proceso muy costoso computacionalmente, por lo que esta búsqueda en lo relativo a **HOG** se ha realizado con 200 imágenes usando el algoritmo **RandomSearch** de *Scikit-Learn*. Las principales ventajas de este algoritmo son:

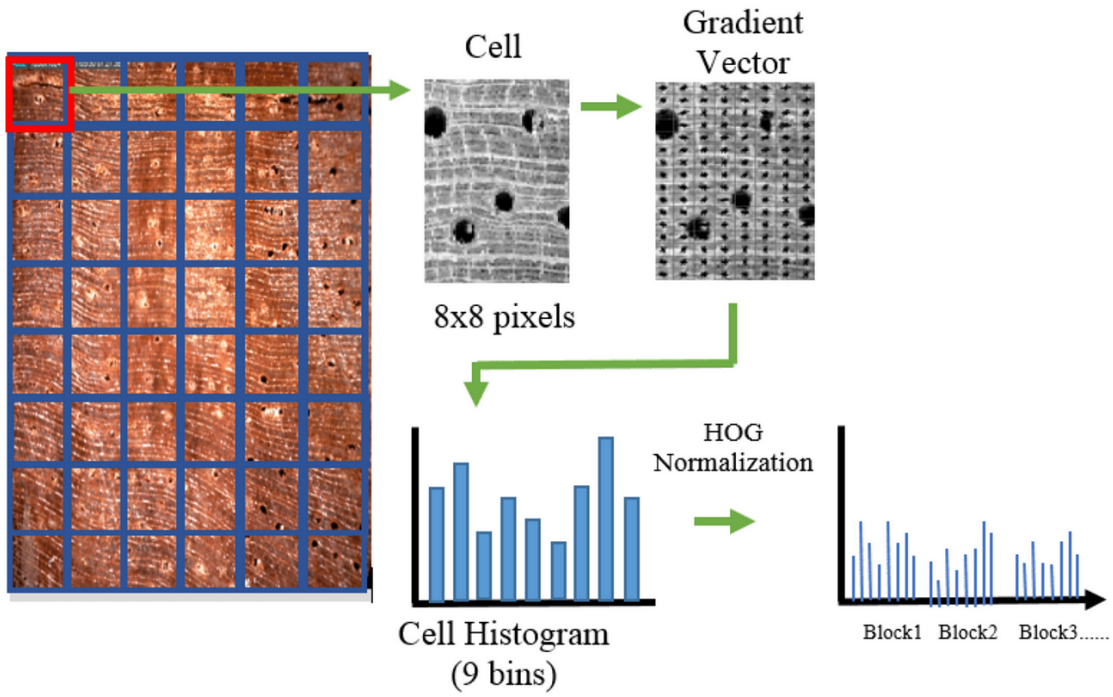

Figura 2: Flujo de cómputo del descriptor **HOG**.

- Un espacio de búsqueda más amplio al realizar combinaciones aleatorias.
- Más eficiente en espacios de dimensionalidad alta.

Los parámetros optimizados no han sido todos. Inicialmente se realizó un estudio de los parámetros tanto experimental como teórico. Dado ese primer paso se decidió que `winSize` sería del tamaño de la imagen, `blockSize` la mitad y `blockStride` y `cellSize` un cuarto. Se realizan búsquedas sobre los siguientes parámetros:

- `nbins`: Número de histogramas por celda. Valores: $\{6, 9, 12\}$.
- `winSigma`: Sigma de la ventana de suavizado gaussiano. Valores: $\{0{,}5, 1{,}0, 2{,}0, 5{,}0\}$.
- `L2HysThreshold`: Umbral para la normalización L2. Valores: $\{0{,}1, 0{,}2, 0{,}3, 0{,}4\}$.
- `signedGradients`: Indicador de gradientes firmados (booleano).
- `gammaCorrection`: Aplicación de corrección gamma (booleano).

Los valores escogidos finalmente para los parámetros tanto del descriptor como para el **SVM** se encuentran descritos en las tablas 2,1.

Parámetro	Valor
svm_kernel	rbf
svm_gamma	1
svm_C	1

Cuadro 1: Parámetros del modelo **SVM**.

Parámetro	Valor
winSize	(28, 28)
blockSize	(14, 14)
blockStride	(7, 7)
cellSize	(7, 7)
nbins	12
winSigma	5
L2HysThreshold	0.3
signedGradients	True
gammaCorrection	0

Cuadro 2: Parámetros del descriptor **HOG**.

4.3. Resultados

Tras buscar los hierparámetros para **HOG** se realizó el ajuste final con un total de 8000 imágenes utilizando una búsqueda de hiperparámetros solo para **SVM** que a su vez por la naturaleza de la implementación, sirve como *k-fold cross validation.*

K-fold cross validation (fig 3) es una técnica utilizada en aprendizaje automático para evaluar el rendimiento de un modelo de manera más confiable. En lugar de dividir los datos en un único conjunto de entrenamiento y prueba, esta metodología divide el conjunto completo en k subconjuntos o *folds*. Posteriormente, el modelo se entrena y evalúa k veces, asegurando que en cada iteración uno de los subconjuntos actúe como conjunto de prueba mientras los restantes se utilizan para el entrenamiento. Este proceso se repite tantas veces como folds se hayan definido, rotando el subconjunto que se usa para la evaluación.

Una vez completadas todas las iteraciones, se calcula el promedio de las métricas de desempeño obtenidas en cada ciclo. Este promedio proporciona una estimación más robusta del rendimiento del modelo, ya que considera variaciones en los datos al usar diferentes particiones para el entrenamiento y la prueba.

Como se puede ver en las tablas 3, 4, los resultados son bastante buenos teniendo en cuenta la dificultad del *dataset* y que solo se ha empleado un descriptor.

Las métricas están muy balanceadas para ambas clases, aunque era de esperar teniendo en cuenta que el conjunto de datos lo está de igual forma.

El área bajo la curva *ROC* (fig 4) sugiere que el modelo tiene un buen desempeño separando las clases, con un 78,95 % de probabilidad de clasificar correctamente una

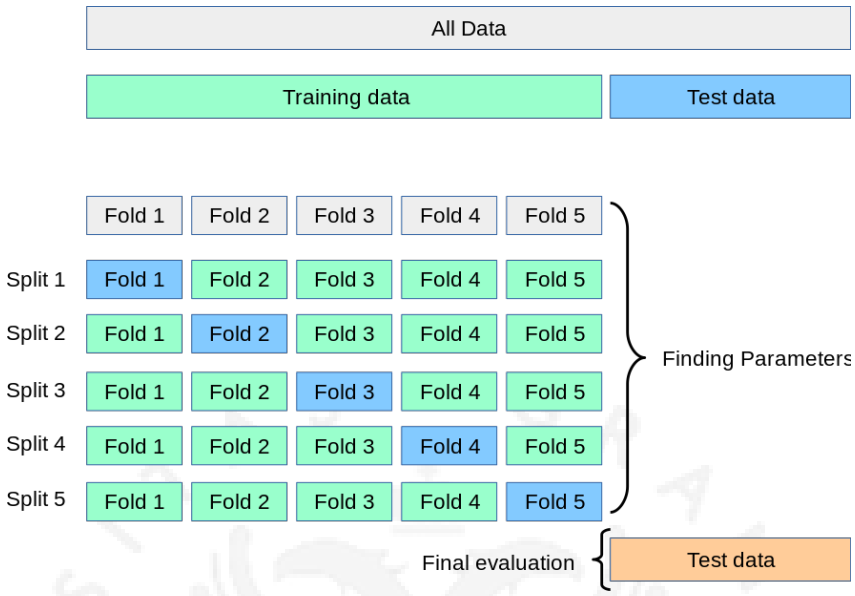

Figura 3: Ejemplo de *k-fold cross validation*.

Metric	Value
Accuracy	0.7131
Precision	0.6976
Recall	0.7408
F1 Score	0.7186
ROC AUC	0.7895
PR AUC	0.7797

Cuadro 3: Métricas de clasificación para **HOG**.

Class	Precision	Recall	F1-Score	Support
0	0.73	0.69	0.71	809
1	0.70	0.74	0.72	791
Macro Avg	0.71	0.71	0.71	1600
Weighted Avg	0.71	0.71	0.71	1600

Cuadro 4: Reporte de clasificación para **HOG**.

muestra positiva frente a una negativa.

El modelo tiene un buen rendimiento con métricas consistentes, aunque hay margen de mejora en la precisión y el área bajo las curvas para aplicaciones críticas, aunque en esos casos es conveniente usar técnicas más complejas.

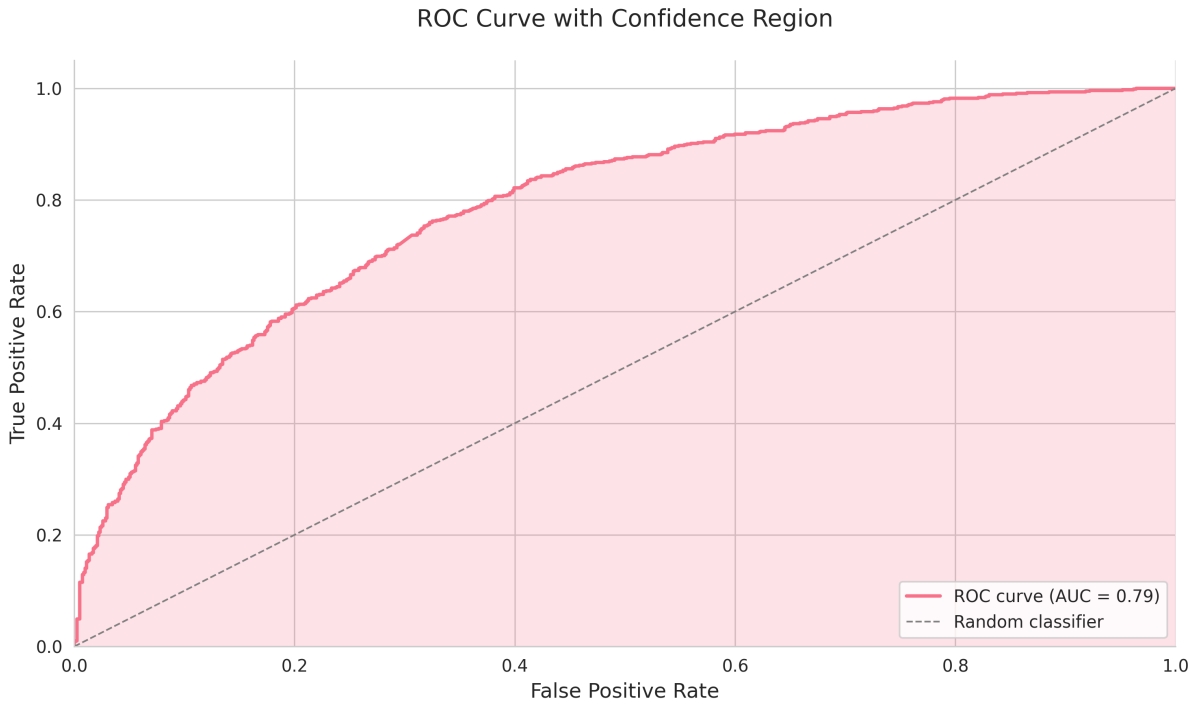

Figura 4: Curva de *ROC* para el modelo **SVM+HOG**.

Los ejemplo de pruebas 5,6,7 han sido clasificados de forma correcta, excepto el último que ha dado un resultado incorrecto.

5. Clasificación con LBP

5.1. LBP

El descriptor **LBP** se basa en analizar la relación de un píxel central con sus píxeles vecinos en una ventana local, capturando patrones que describen la estructura de la textura de la región.

Se compara la intensidad del píxel central con cada uno de los píxeles vecinos. Si la intensidad de un vecino es mayor o igual a la del píxel central, se asigna un valor de uno, de lo contrario, se asigna un valor de cero. Esto produce un conjunto de valores binarios, que se organizan como un número binario. Al convertir este número binario a su equivalente decimal, se obtiene el valor LBP correspondiente al píxel central.

Figura 5: Ejemplo de prueba 1.

Figura 6: Ejemplo de prueba 2.

Figura 7: Ejemplo de prueba 3.

El cálculo se realiza iterando a través de toda la imagen en una ventana deslizante, generando un mapa de características donde cada píxel está representado por su valor LBP. Este mapa puede interpretarse como una nueva representación de la imagen, enfatizando los patrones locales de textura.

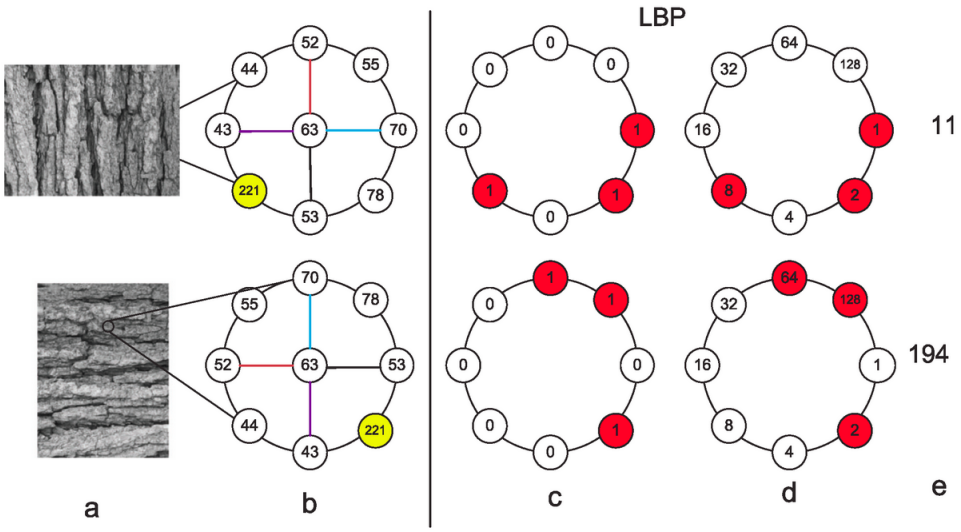

Figura 8: Ejemplo de cálculo de **LBP**.

5.2. Implementación

Se ha realizado una implementación propia del descriptor, siguiendo la interfaz de **HOG** en *OpenCV* para mayor coherencia y cohesión con el *software* desarrollado.

La función de inicialización se encarga de recoger los paráemtros y realizar ciertas comprobaciones necesarias para el correcto funcionamiento del algoritmo, de esta forma el usuario será advertido si no los está introduciendo correctamente. Se comprueban cosas básicas como que el `radius` (radio) no pueda ser menor que 1 y que `n_neighbors` (número de vecinos) sean múltiplos de 8 y nunca menor a ese número.

```
    def __init__(self, radius: int = 1, n_neighbors: int = 8):
        if radius < 1:
            raise ValueError("The radius must be greater than 0.")
        if n_neighbors % 8 != 0 or n_neighbors < 8:
            raise ValueError(
                "The number of neighbors (n_neighbors) must be a multiple of 8 and at least 8."
            )
        self.radius = radius
        self.n_neighbors = n_neighbors

        # Precompute relative offsets for neighbors
        self.neighbor_offsets = [
            (
```

```
                radius * np.cos(2 * np.pi * i / n_neighbors),
                radius * np.sin(2 * np.pi * i / n_neighbors),
            )
            for i in range(n_neighbors)
        ]
```

La función principal del algoritmo es la de `compute` que se encarga de la funcionalidad principal de codificación de la imagen. De nuevo realiza comprobaciones básicas, y si se pasan, calcula el patrón binario asociado a cada píxel con sus vecinos dentro de un radio específico. Normalmente es necesario realizar una interpolación de los píxeles de los vecinos, pues se calculan siguiendo una fórmula radial.

Después de obtener el patrón es necesario pasar este a decimal y, finalmente, calcular el histograma.

```
    def compute(self, img: np.ndarray) -> np.ndarray:
        """
        Computes the Local Binary Pattern (LBP) for the given image.

        Args:
            img (np.ndarray): Grayscale image as a 2D numpy array.
        Returns:
            np.ndarray: Flattened normalized histogram of LBP values.
        """
        if len(img.shape) != 2:
            raise ValueError("Input image must be a 2D grayscale image.
")

        rows, cols = img.shape
        lbps = np.zeros(
            (rows - 2 * self.radius, cols - 2 * self.radius), dtype=np.
uint8
        )

        for i in range(self.radius, rows - self.radius):
            for j in range(self.radius, cols - self.radius):
                center_pixel = img[i, j]
                binary_pattern = 0

                for idx, (dx, dy) in enumerate(self.neighbor_offsets):
                    neighbor_value = self._bilinear_interpolation(img,
i + dy, j + dx)
                    binary_pattern |= (neighbor_value > center_pixel)
<< idx

                lbps[i - self.radius, j - self.radius] = binary_pattern

        # Compute histogram of LBP values
        hist, _ = np.histogram(lbps.flatten(), bins=np.arange(255),
range=(0, 255))
        return hist / hist.sum()
```

El cálculo de los vecinos se realiza mediante la siguiente fórmula. Esto sol:

$$x'_i = r \cdot \cos\left(\frac{2\pi i}{n}\right) + x_i, \quad y'_i = r \cdot \sin\left(\frac{2\pi i}{n}\right) + y_i$$

Otra opción posible habría sido coger el vecino más cercano en vez de interpolar.

Se ha creado un diagrama (fig 9) que describe las funcionalidades de la clase y cómo se ha dividido el trabajo en distintos métodos dentro de esta.

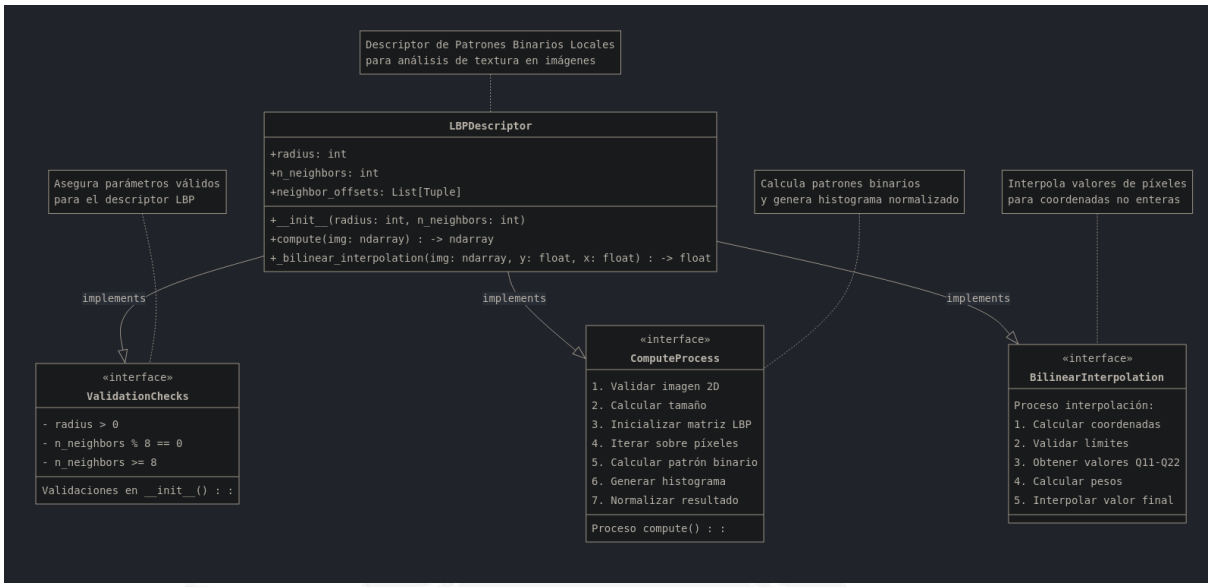

Figura 9: Diagrama de la clase **LBP**.

5.3. Resultados

Metric	Value
Accuracy	0.6469
Precision	0.6549
Recall	0.6731
F1 Score	0.6639
ROC AUC	0.7042
PR AUC	0.6960

Cuadro 5: Métricas de clasificación para **LBP**.

Como puede observarse en las figuras 6, 5, las métricas obtenidas son igual de balanceadas que las de **SVM+HOG**. En este caso la predicción es menos potente en todos los sentidos, los resultados son inferiores y el tiempo de cómputo (al ser una implementación en *Python*) más elevado.

Los ejemplo de pruebas 5,6,7 han salido peor parados que con **HOG**. El primero correcto y los otros dos incorrectos.

Class	Precision	Recall	F1-Score	Support
0	0.64	0.62	0.63	771
1	0.65	0.67	0.66	829
Macro Avg	0.65	0.65	0.65	1600
Weighted Avg	0.65	0.65	0.65	1600

Cuadro 6: Reporte de clasificación para **LBP**.

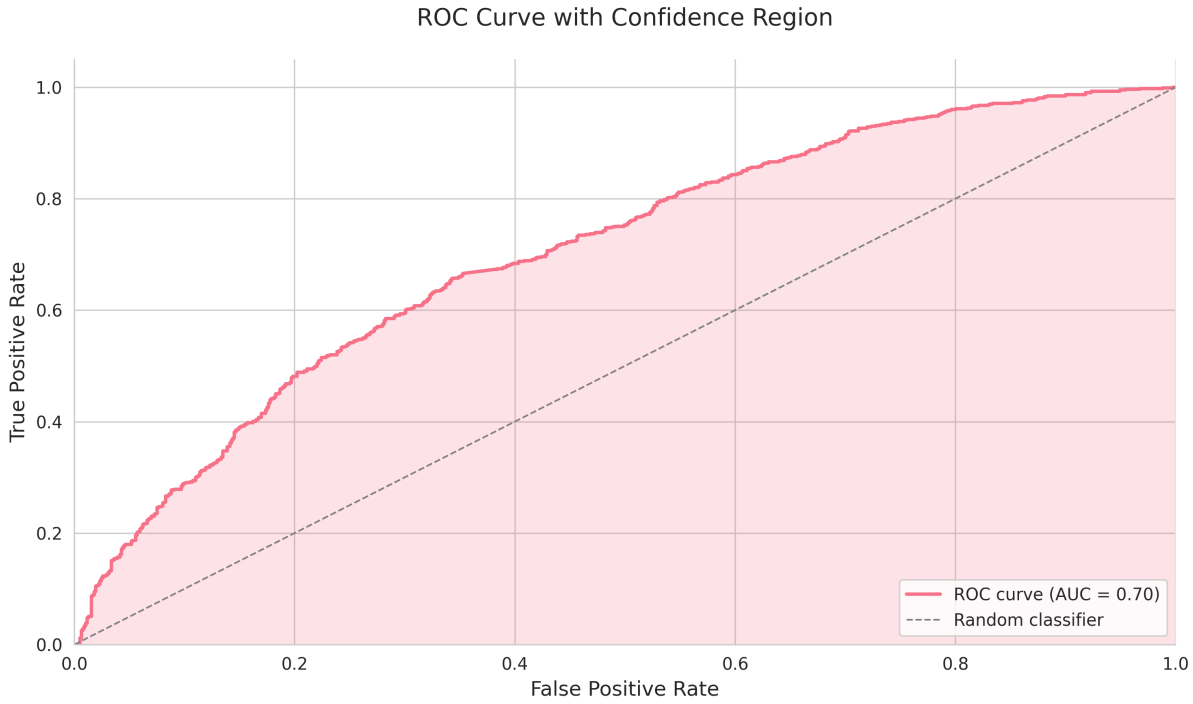

Figura 10: Curva de *ROC* para el modelo **SVM+LBP**.

La curva *ROC* (fig 10) obtenida es bastante menos pronunciada que la anterior, más plana, pero aún así son resultados decentes.

6. Clasificación con tres clases

Para la clasificación con otra clase extra se han añadido 2700 imágenes de caballos, de forma que ahora el predictor debe decidir entre gatos, perros y caballos. Estas imágenes han sido sacadas del siguiente enlace: *https://www.kaggle.com/datasets/alessiocorrado99/animals10.*

Se han recortado las imágenes de gatos y perros para poder equilibrar el número de imágenes por clase. De todas formas, son bastantes.

6.1. Resultados con HOG

Parámetro	**Valor**
svm_kernel	rbf
svm_gamma	0.1
svm_C	1

Cuadro 7: Parámetros del modelo **SVM** con el descriptor **HOG**.

Los parámetros encontrados en la búsqueda de hiperparámetros para **SVM** son los mencionados en la figura 7.

Como puede verse en la figura 11, los resultados son bastante buenos. Añadiendo una clase sigue siendo capaz de diferenciar las tres. Caballo es mucho más diferenciable que un gato o perro, como puede verse en el gráfico.

Metric	**Value**
Accuracy	0.7081
Precision	0.7081
Recall	0.7081
F1 Score	0.7081

Cuadro 8: Métricas de clasificación para el modelo con **HOG** con tres clases.

En las tablas 8, 9 quedan registrados los resultados del clasificador. La clase caballo es mucho más diferenciable que el resto. Pese a ello, se obtienen resultados buenos.

6.2. Resultados con LBP

Los parámetros encontrados en la búsqueda de hiperparámetros para **SVM** son los mencionados en la figura 10

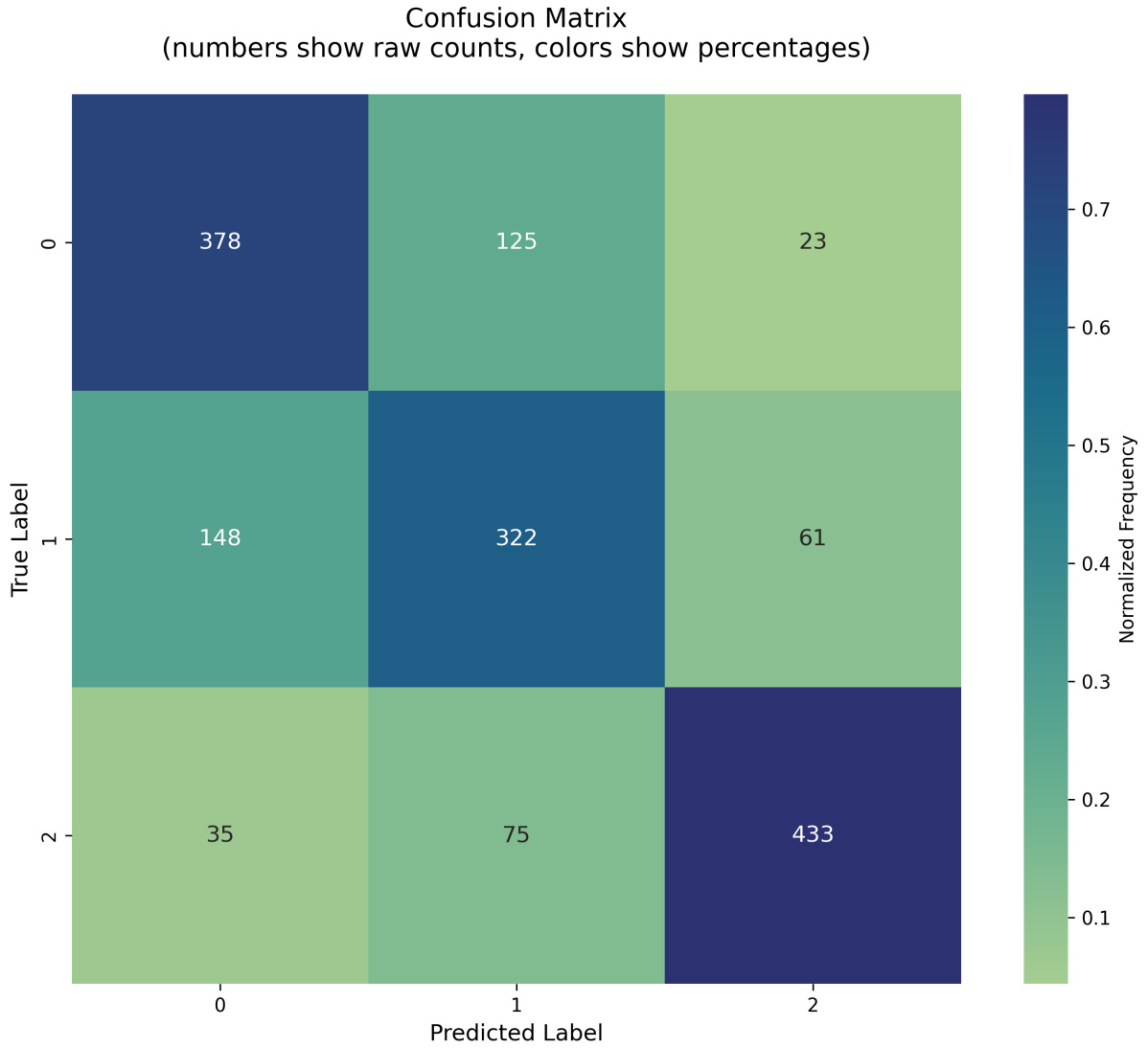

Figura 11: Matriz de confusión con **SVM+HOG** para tres clases.

Class	Precision	Recall	F1-Score	Support
0	0.67	0.72	0.70	526
1	0.62	0.61	0.61	531
2	0.84	0.80	0.82	543
Accuracy		0.71 (total: 1600)		
Macro Avg	0.71	0.71	0.71	1600
Weighted Avg	0.71	0.71	0.71	1600

Cuadro 9: Reporte de clasificación para el modelo con **HOG** con tres clases.

Parámetro	Valor
svm_kernel	rbf
svm_gamma	0.1
svm_C	1

Cuadro 10: Parámetros del modelo **SVM** con el descriptor **LBP**.

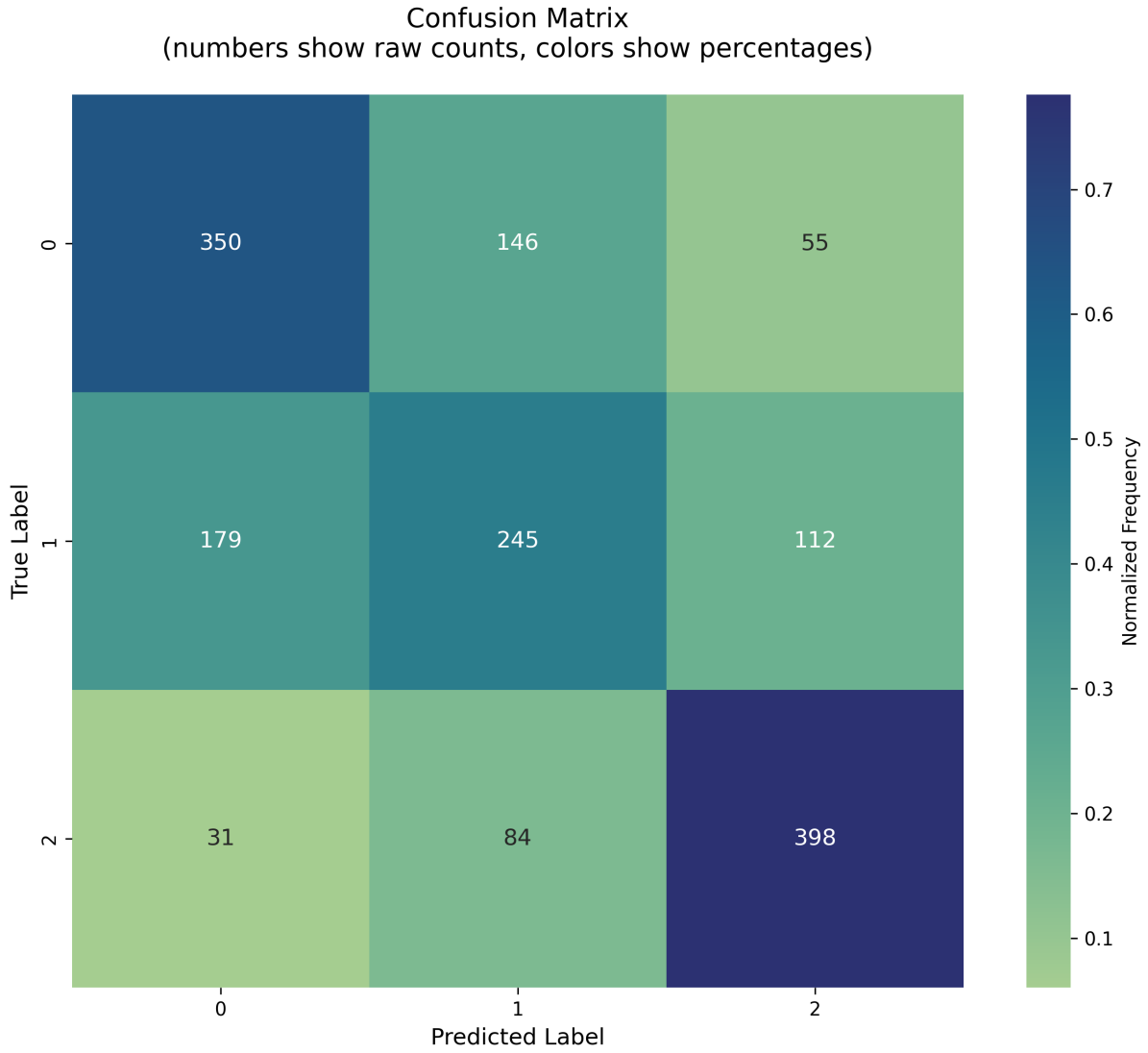

Figura 12: Matriz de confusión con **SVM+LBP** para tres clases.

Metric	Value
Accuracy	0.6206
Precision	0.6206
Recall	0.6206
F1 Score	0.6206

Cuadro 11: Métricas de clasificación para el modelo con **LBP** para tres clases.

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.62	0.64	0.63	551
1	0.52	0.46	0.48	536
2	0.70	0.78	0.74	513
Accuracy		0.6206 (total: 1600)		
Macro Avg	0.62	0.62	0.62	1600
Weighted Avg	0.61	0.62	0.62	1600

Cuadro 12: Reporte de clasificación para el modelo con **LBP** para tres clases.

Los resultados mostrados por **LBP** en las figuras 12 y tablas 12, 11 son exáctamente iguales que en **HOG**, pero con una precisión peor.

Dados estos resultados es visible que el descriptor **LBP** parece menos potente en este problema comparándolo con **HOG**.