Práctica 1

Aprendizaje Automático

Àngel Jiménez Sanchis
Albert Salom Vanrell

CONTEXTUALIZACIÓN

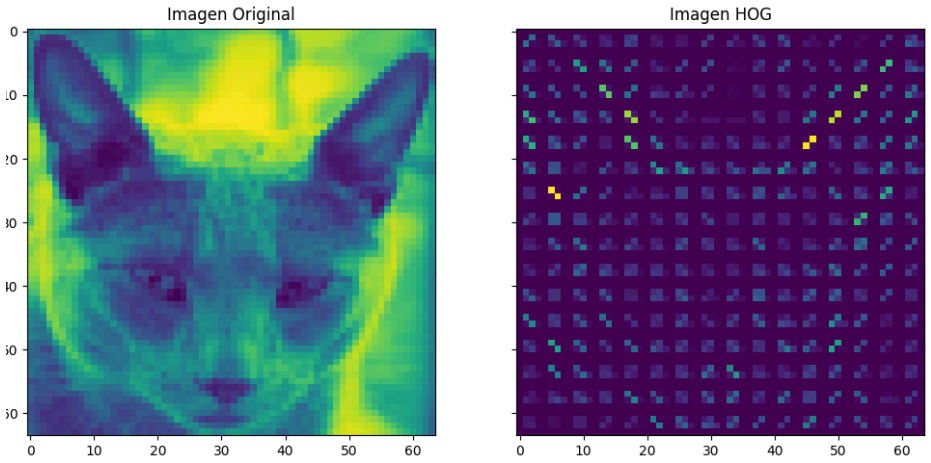
Para comenzar, utilizaremos un dataset de imágenes de perros y gatos titulado 'Dog and Cat Detection' de Kaggle. Este conjunto de datos incluye imágenes etiquetadas para indicar si corresponde a un perro o a un gato. Además, las imágenes estarán recortadas, mostrando únicamente la cabeza de cada animal.

HOG

Una vez cargado el dataset, procederemos a extraer las características de cada foto para luego proceder a entrenar el modelo. Para realizar el HoG, primero hay que definir la configuración de este.

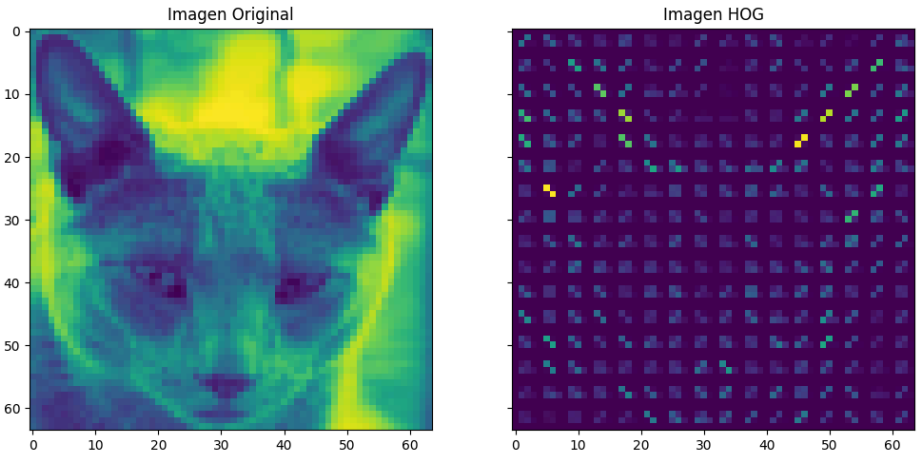
Esto se ha llevado a cabo mediante un método llamado "*configuracionHoG()*", donde se evaluaron las diferentes configuraciones que establecimos, recorriendo inicialmente un conjunto de 6 imágenes para establecer la configuración adecuada observando cada HoG mediante los plots.

Pixels per block: 4x4, cells per block: 2x2 y orientations: 9



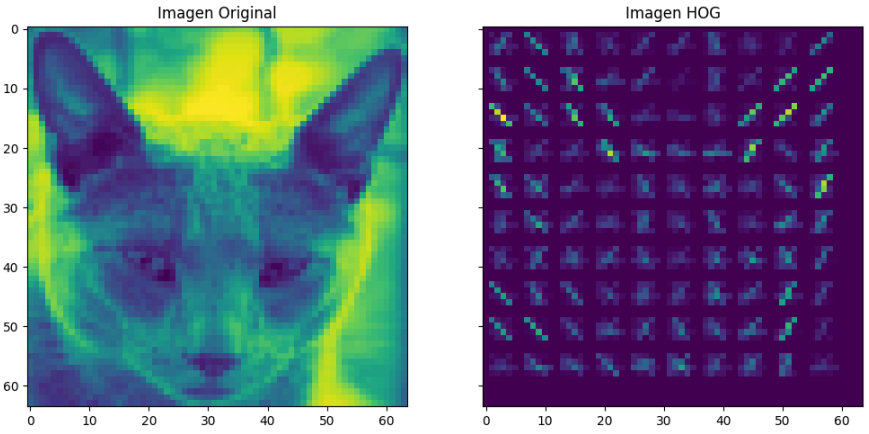
El hecho de tener los píxeles por celda de 4, hace que las características que se obtienen sean muy detalladas, provocando que en imágenes con fondos más complejos capture mucho ruido o detalles innecesarios, que podrían empeorar nuestra clasificación y provocar overfitting.

Pixels per block: 4x4, cells per block: 8x8 y orientations: 9



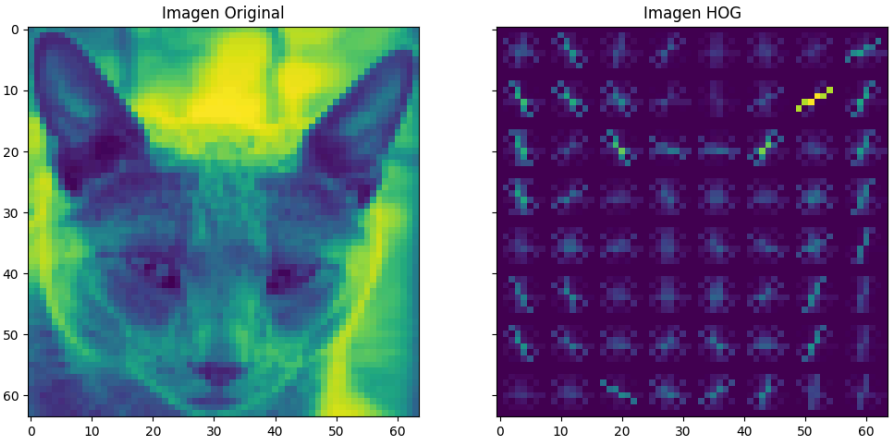
Como podemos observar, no se encuentran diferencias visuales entre este hog y el anterior. Esto se puede deber a que, como trabajamos con imágenes de 64x64 píxeles, puede que no sean suficientes para poder encontrar diferencias si variamos el cells per block.

Pixels per block: 6x6, cells per block: 2x2 y orientations: 9



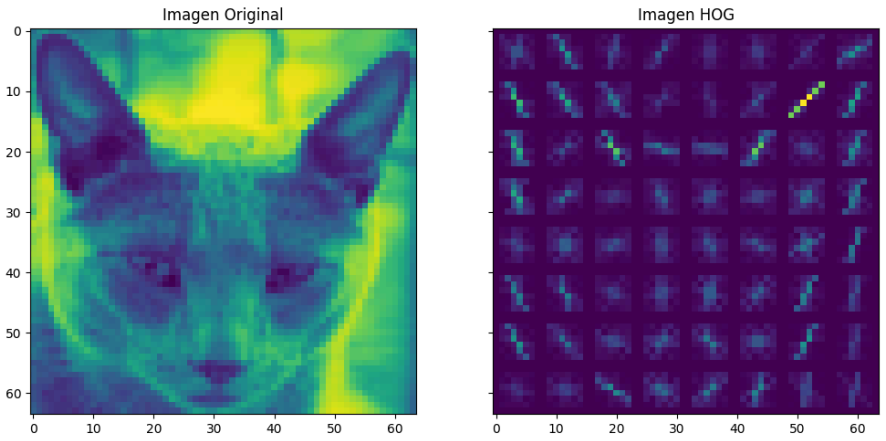
Esta es una configuración intermedia en cuanto al valor de píxeles por celda, pero quizá para nuestro tamaño de imagen, sigue capturando bastante detalle que no necesitamos, así que preferimos tener un valor más elevado.

Pixels per block: 8x8, cells per block: 2x2 y orientations: 9



Casi siendo idéntica a la configuración que trae el HoG por defecto, esta es la que mejor parece ajustarse a nuestras imágenes, porque permite capturar bien el detalle sin llegar a ser demasiado sensible a ellos. Los bloques son de un tamaño una unidad inferior al valor por defecto ya que nuestras imágenes son relativamente pequeñas y la cantidad de orientaciones nos ofrece bastante detalle sin llegar a ser excesivo o muy complejo.

Pixels per block: 8x8, cells per block: 2x2 y orientations: 18



El hecho de cambiar las orientaciones a 18, podría ser muy útil para capturar bien el detalle pero para nuestro caso, esto provocaría una mayor cantidad de características que posiblemente podrían ser más de las que necesitamos sin notar realmente una gran mejoría.

Teniendo todo esto en cuenta, nos decantaremos por usar la configuración de 8 pixels per block, 2 cells per block y 9 orientaciones, ya que consideramos que es

la más adecuada para realizar este estudio y clasificar correctamente tanto a gatos como a perros.

Con esta configuración extraeremos las características de todo el Dataset, que serán almacenadas en un archivo, para facilitar su carga.

Es importante también destacar la diferencia en cuanto al fondo de las imágenes del Dataset, ya que hay que disponen de fondos simples que permiten que el HoG capture muy bien las características relevantes de la imagen mientras que otras tienen fondos muy cargados o con mucho nivel de detalle que puede confundir a nuestro clasificador introduciendo gradientes irrelevantes que se podrían confundir con características importantes.

SVM Y BÚSQUEDA EXHAUSTIVA

Una vez obtenido el HoG con la configuración seleccionada, se procede a cargar esas características almacenadas en un archivo, junto con las etiquetas que nos indican si la muestra es un gato '0' o un perro '1' y se pasan como parámetros a la función "train_test_split()" que se encargará de dividir nuestros datos en entrenamiento y test.

Una vez separados, se estandarizan usando el "StandardScaler()" y establecemos un diccionario de parámetros, "params_kernels", que contiene los parámetros de configuración para diferentes tipos de kernel.

```
# Estandarización de los datos:
scaler = StandardScaler()
X_transformed = scaler.fit_transform(X_train)
X_test_transformed = scaler.transform(X_test)
```

A continuación, usamos un bucle para iterar sobre cada tipo de kernel en param_kernels, se crea un modelo SVM y se optimiza usando "GridSearchCV()", que es el que se encarga de realizar una búsqueda exhaustiva explorando todas las combinaciones posibles de los parámetros indicados para cada kernel y selecciona la mejor configuración en función de su rendimiento en validación cruzada (cv=5) y se entrena el modelo para cada configuración.

Una vez finalizada la búsqueda, el modelo con la mejor combinación de parámetros se guarda en el diccionario best_models usando grid_search.best_estimator_, que devuelve el modelo SVM ajustado con los mejores parámetros encontrados para el kernel actual.

Además de esto, se realiza una predicción con cada kernel y su mejor modelo encontrado, evaluando el "accuracy" para ver si tiene un buen rendimiento.

DISCUSIÓN PARÁMETROS

RBF

```
'kernel': 'rbf', 'C': 3, 'gamma': 'scale', 'tol': 1, 'max_iter': 2000
```

En el modelo RBF, observamos que los mejores resultados se lograron con valores de C superiores a 1, lo cual sugiere que el valor por defecto producía underfitting. Además, al relajar la tolerancia por encima del valor predeterminado (1e-3) y fijar un máximo de 2000 iteraciones, logramos optimizar la precisión sin incrementar excesivamente el tiempo de procesamiento. Esto indica que el aumento de `max_iter` no mejoraba la precisión de manera significativa, mientras que los tiempos de cálculo aumentaban exponencialmente.

Lineal

```
'kernel': 'linear', 'C': 0.0001, 'tol': 1, 'max_iter': 500
```

En este caso, vemos una C muy inferior al rbf, esto puede ser debido a que este modelo es más simple y los datos están más regularizados. Por otro lado, vemos que hemos podido acotar aún más el número máximo de interacciones, esto puede ser debido a la simplicidad del modelo y al rápido aprendizaje del modelo.

Polinómico

```
'kernel': 'poly', 'C': 1, 'degree': 2, 'gamma': 'auto', 'coef0': 0.1,
'tol': 0.5, 'max_iter': 1500
```

Podemos observar, que el polinómico con mejores resultados es de 2° grado. Con una C=1, vemos que no penaliza demasiado los errores como el rbf, pero que tampoco es tan laxo como el lineal. En cuanto al coef0, observamos que es mejor darle menos peso a los valores de primer grado y más a los de 2° grado. En cuanto a la tolerancia, vemos que es ligeramente menor, pero demasiado por encima del valor por defecto.

RESULTADOS OBTENIDOS

Una vez que se han obtenido los mejores parámetros para cada kernel, se los pasamos a la función “train_and_evaluate_fix_model()”, junto con los datos de train y test, para realizar las evaluaciones pertinentes.

Primero de todo, cargamos un SVM con esos parámetros y le realizamos el entrenamiento y la predicción.

Para extraer los resultados de precisión, recall, f1-score y accuracy usamos la función “calcular_metricas()” que recibe como parámetros la y_real y la y_predicha por nuestro modelo.

Además de comprobar esto, también nos pareció adecuado mirar si el modelo estaba sufriendo de overfitting o underfitting, para ello, simplemente realizamos la predicción con el conjunto de train, calculamos su accuracy y miramos que los resultados de accuracy y test, si la diferencia que hay entre ellos es pequeña, y su accuracy en ambos casos es alto, significa que nuestro modelo funciona correctamente.

TABLA COMPARATIVA Y GRÁFICA

Primeramente, hay que destacar que nos pareció conveniente calcular cuántas imágenes teníamos de cada tipo en nuestro conjunto de test, para saber si había mucha diferencia entre clases y comprender si sería necesario aplicar un parámetro de balanceo de clases en las métricas.

Esto se ha llevado a cabo con la función “contar_muestras()” que recibe las etiquetas de todo nuestro conjunto de test y va identificando cada una de ellas a qué clase pertenece.

```
Número de muestras por clase (0 → Gato, 1 → Perro):
Clase 0: 252
Clase 1: 486
```

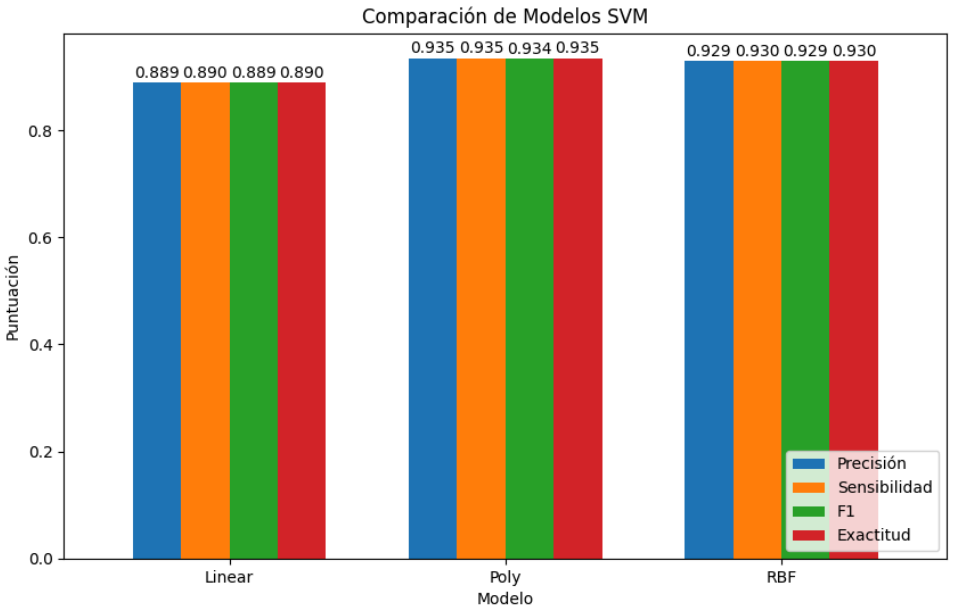
Como podemos observar hay una diferencia notoria entre ellas, por tanto sí será importante aplicar este parámetro.

Dicho esto, para mostrar correctamente los resultados de las métricas para cada tipo de kernel, cuando estamos evaluando los distintos modelos en nuestro main, almacenamos esos valores de las métricas en una lista, que después se le pasa a una función “construir_tabla_y_grafico()”.

Dentro de esta función, se crea un dataFrame con las métricas obtenidas, y se muestra adecuadamente.

	Modelo	Precisión	Sensibilidad	F1	Exactitud
0	Linear	0.889368	0.890244	0.889146	0.890244
1	Poly	0.934872	0.934959	0.934420	0.934959
2	RBF	0.929229	0.929539	0.929110	0.929539

Además de la tabla comparativa, también aprovechamos para construir una gráfica que es más visual.



DISCUSIÓN RESULTADOS

Observando la gráfica puede apreciarse que el **kernel lineal** obtuvo una **puntuación de 0.89** en todas las métricas (debido a aplicar el escalado de las clases). Esto sugiere que el clasificador SVM con kernel lineal es consistente en su capacidad de identificar correctamente gatos y perros, aunque con una resultados un poco menores que los otros kernels.

Debido a esta diferencia en los resultados obtenidos, aunque sea pequeña, podría servir para indicarnos que nuestras imágenes y por tanto, nuestras características obtenidas por el HoG, no tienen una separación completamente lineal, por lo que se le dificulta un poco más capturar algunas características.

El modelo con **kernel polinómico** alcanzó una **puntuación de 0.935** en todas las métricas. Esto nos indica que el kernel polinómico logra captar patrones no lineales presentes en los datos, modelando mejor la diferencia entre gatos y perros.

El modelo con **kernel RBF** también alcanzó una **puntuación de 0.93** en todas las métricas, casi igual que el kernel polinómico. Esto nos sugiere que ambos modelos capturan mejor algunos patrones no lineales en nuestros datos, y por eso rinden mejor que el modelo con kernel lineal.

La elección entre el kernel polinómico y el RBF puede depender de factores como la complejidad y el tiempo de entrenamiento. El RBF suele ser más exigente con los recursos, ya que necesita modelar fronteras de decisión más complejas.

```
Tiempo de ejecución del modelo linear: 3.931 segundos
Tiempo de ejecución del modelo poly: 4.309 segundos
Tiempo de ejecución del modelo rbf: 5.166 segundos
```

Como podemos observar, el tiempo en este caso es un factor importante en la decisión del mejor modelo. El modelo polinómico, además de tener unas mejores puntuaciones, es más rápido que el rbf. Por tanto, el modelo polinomial sería el mejor modelo para este dataset.

Por otro lado, aunque el kernel lineal tiene un rendimiento un poco más bajo, es menos complejo y requiere menos recursos, podría ser útil en casos donde se valora la simplicidad y rapidez antes que el máximo rendimiento.

CLASIFICACIÓN DE IMÁGENES

Como podemos observar, el modelo funciona mejor en las fotos donde el animal está mirando hacia la cámara o, en el caso de los perros, cuando tienen la boca abierta. Esto se debe a que, con la boca abierta, es más fácil diferenciar la forma

distintiva de la mandíbula entre perros y gatos. Además, algunas razas de perro tienen orejas caídas, una característica que no se observa en los gatos, lo que permite al modelo acertar con mayor facilidad en estos casos, ya sea que el animal esté de frente o de lado.

En cuanto a los errores, el modelo tiende a confundir a los perros con pelaje corto y orejas puntiagudas, características que son más comunes en los gatos. También se observa que algunos animales en posiciones inclinadas frente a la cámara dificultan que el modelo aprecie bien sus rasgos, lo que provoca confusiones en la clasificación.