

Laboratorio Nº 7- Aprendizaje Automático en Visión Artificial			
Asignatura	Tópicos Especiales - Visión Artificial	Código	0756
Profesor	José Carlos Rangel Ortiz		

Introducción

Siguiendo con el módulo de Machine Learning, dentro de este podemos encontrar el Deep Learning, el cual se fundamenta en la utilización de Redes Neuronales Artificiales con una gran cantidad de capas para el procesamiento y aprendizaje de la información que se suministre.

En este laboratorio se presentarán formas de construir modelos de clasificación utilizando una varios algoritmos de ML. Durante esta parte del laboratorio el modelo será entrenado utilizando datos (imágenes) que se encuentran localmente en nuestra computadora, o dicho de otra manera, un dataset local. El laboratorio muestra todo el procedimiento y permite también evaluar el modelo generado utilizando una imagen externa a las disponibles en el dataset.

El laboratorio puede ser desarrollado construyendo el *script* completamente y ejecutando desde consola o mediante *jupyter lab* realizando una ejecución por celdas.

Objetivo

Aplicar los diferentes conceptos relacionados al aprendizaje automático enfocado en problemas de visión artificial.

Entregables

Al final de cada sección del laboratorio se indicará cuales son acciones que deberá cumplir el estudiante y los cuales serán evaluados en la actividad. El informe final debe contener todo lo solicitado y se debe redactar utilizando la Guía de Estilos para Memorias y Laboratorios Disponible en Moodle. El informe debe ser entregado en formato PDF y siguiendo las indicaciones para nombrar el archivo.

1. Support Vector Machines SVM - usando OpenCV

Dentro de los algoritmos de ML que se utilizan mayormente, podemos mencionar SVM, el cual durante muchos años previos a las RRNN se posicionó como el principal método para construir clasificadores para diversos tipos de problemas en visión artificial. En la actualidad sigue siendo considerado como un clasificador que produce buenos resultados para algunos problemas. Los ejemplos de esta sección se han extraído del libro *Mastering OpenCV 4 with Python*^[1]. En esta sección el archivo de código se llama: `01_svm_handwritten_digits_recognition_preprocessing_hog.py`.

En esta sección se utilizará el dataset Digits, que esta conformado por 5000 imágenes divididas en 10 clases diferentes. Este dataset contiene imágenes de los 10 dígitos (0 – 9) escritos a mano.

Para este dataset OpenCV dispone de una imagen con resolución de 2000×1000 y en dicha imagen en bloques de 20×20 se encuentran las 5000 imágenes que conforman el dataset, la imagen completa se muestra en la Figura 1, mientras que la Figura 2 incluye un extracto de los elementos disponibles.

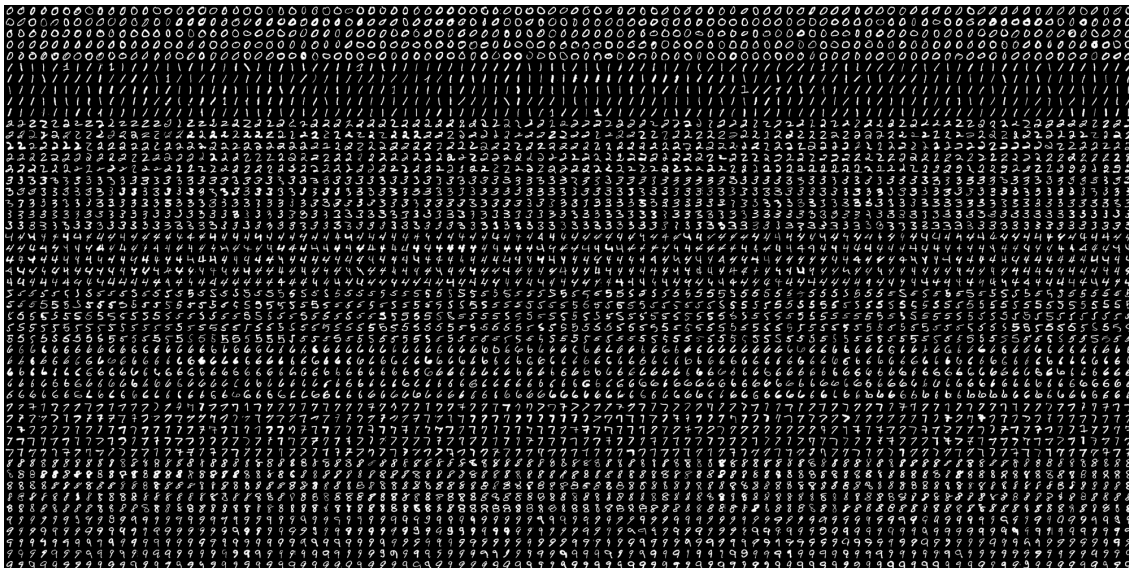


Figura 1: Dataset DIGITS

- Tomando en cuenta lo anterior las imágenes deben ser recortadas y pre-procesadas. Para recortar los datos y definir la etiqueta de cada imagen en el dataset se hace uso de la función `load_digits_and_labels()`.

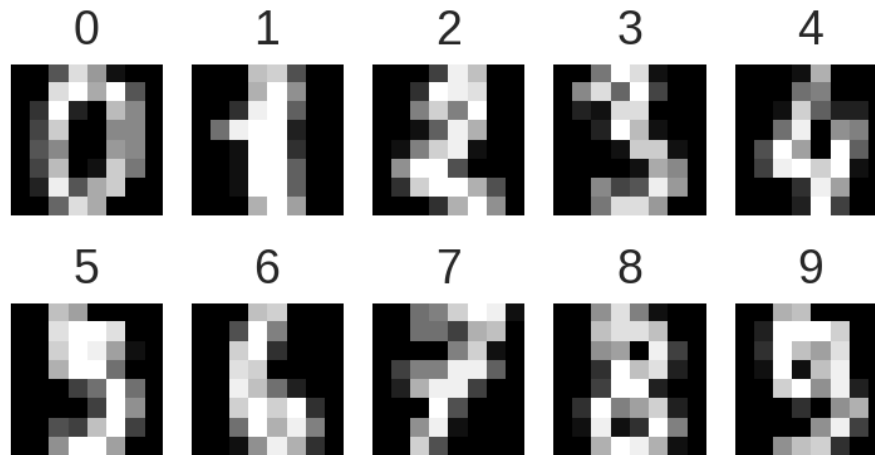


Figura 2: Muestra del Dataset DIGITS

```
def load_digits_and_labels(big_image):
    """ Returns all the digits from the 'big'
    image and creates the corresponding labels for each image"""

    # Load the 'big' image containing all the digits:
    digits_img = cv2.imread(big_image, 0)

    # Get all the digit images from the 'big' image:
    number_rows = digits_img.shape[1] / SIZE_IMAGE
    rows = np.vsplit(digits_img, digits_img.shape[0] / SIZE_IMAGE)

    digits = []
    for row in rows:
        row_cells = np.hsplit(row, number_rows)
        for digit in row_cells:
            digits.append(digit)
    digits = np.array(digits)

    # Create the labels for each image:
    labels = np.repeat(np.arange(NUMBER_CLASSES), len(digits) / NUMBER_CLASSES)
    return digits, labels
```

- De la misma manera la función `deskew()` se encarga de eliminar los sesgos a las imágenes y producir una mejor representación de los datos para entrenar el clasificador.

```
def deskew(img):
    """Pre-processing of the images"""
```

```

m = cv2.moments(img)
if abs(m['mu02']) < 1e-2:
    return img.copy()
skew = m['mu11'] / m['mu02']
M = np.float32([[1, skew, -0.5 * SIZE_IMAGE * skew], [0, 1, 0]])
img = cv2.warpAffine(img, M,
                    (SIZE_IMAGE, SIZE_IMAGE),
                    flags=cv2.WARP_INVERSE_MAP | cv2.INTER_LINEAR)

return img

```

- Para este tipo de clasificador además de pre-procesar las imágenes, se hace necesario el cálculo de un descriptor global para cada una de las imágenes. Para nuestro clasificador se utilizará el descriptor global (*Histogram of Oriented Gradients*, HoG)[2]. Este procedimiento se lleva a cabo en la siguiente función:

```

# HoG feature descriptor:
hog = get_hog()

# Compute the descriptors for all the images.
# In this case, the HoG descriptor is calculated
hog_descriptors = []
for img in digits:
    hog_descriptors.append(hog.compute(deskew(img)))
hog_descriptors = np.squeeze(hog_descriptors)

```

- Como siguiente punto se divide el dataset en secuencias de *train* y *test*, en este caso de un 70 % para entrenamiento y de 30 % para test.

```

# At this point we split the data
# into training and testing (70% for train):
partition = int(0.7 * len(hog_descriptors))
hog_descriptors_train, hog_descriptors_test = np.split(hog_descriptors, [partition])
labels_train, labels_test = np.split(labels, [partition])

```

- Como siguientes pasos se procede a crear las instancias de nuestro clasificador SVM. Para este ejemplo se utiliza la instancia de SVM incluida en OpenCV. En este caso se utilizan diferentes funciones para facilitar el proceso.
- Las diferentes funciones que permiten crear y utilizar SVM son las siguientes:
 - `svm_init` : Esta función crea la instancia de SVM y define cuales son los parámetros que debe utilizar, estos parámetros permiten la correcta ejecución del modelo y de igual manera su ajuste permitiría mejorar el desempeño del mismo. Los argumentos **C** y **gamma**, son los parámetros que mayor peso tienen en la ejecución del algoritmo. Más información se puede encontrar en esa web¹

¹SVM OpenCV

- `svm_train`: Esta función recibe una instancia de modelo y procede a realizar el entrenamiento del mismo, utilizando los datos proporcionados.
 - `svm_predict`: Esta función recibe un modelo previamente entrenado y una lista de instancias(descriptores). La función envía cada elemento de la lista al modelo y almacena la predicción o salida del modelo para cada instancia.
 - `svm_evaluate`: Esta función recibe un modelo, una lista de instancias y las etiquetas (*ground truth*) de cada instancia. Realiza la predicción para cada instancia y calcula el accuracy para el modelo, con ese conjunto de datos. Devuelve un valor de porcentaje de aciertos.
- Como siguiente punto se procede con el entrenamiento del modelo usando la secuencia de *train* del dataset.

```
print('Training SVM model ...')
model = svm_init(C=12.5, gamma=0.50625)
svm_train(model, hog_descriptors_train, labels_train)
```

- Como último punto debemos probar nuestro modelo utilizando las imágenes de *test*, lo cual se realiza en el siguiente fragmento.

```
print('Evaluating model ... ')
svm_evaluate(model, hog_descriptors_test, labels_test)
```

- Puede también ejecutar el programa:
- `01_svm_handwritten_digits_recognition_preprocessing_hog_c_gamma`.
- Este aplica unos cambios en los parámetros definidos de C y Gamma del algoritmo, para evaluar su efecto en el accuracy obtenido.

Entregables para la Sección

1. Para esta sección genere una matriz de confusión para la secuencia de train y de test creada en el código, utilizando la función `ConfusionMatrixDisplay.from_predictions()` de SKLearn. Puede encontrar la información de esta función en este [Enlace](#).
2. Coloque adecuadamente el titulo a cada matriz y presente la matriz y una captura del código utilizado para generar la matriz en su informe de laboratorio.

2. Support Vector Machines SVM - usando SKLearn

En esta sección se procederá a utilizar SciKit-Learn, la cual es una de las librerías de mayor uso para temas de aprendizaje automático. Es una librería que ha estructurado todos sus clasificadores en bajo una sola clase padre la cual hereda sus atributos a cada uno de los estimadores que se crean en base a esta. En otras palabras SKLearn definió una clase llamada estimadores, en la cual se definen los métodos básicos y comunes para el trabajo con clasificadores en Python. Cada algoritmo de clasificador (Perceptrones, SVM, KNN, K-Means, etc) hereda entonces estos métodos de la clase padre, y solo cambia su ejecución interna. Por lo cual, a pesar de tener métodos con nombres similares, su implementación varia dependiendo del enfoque.

Esta estructuración de estimadores, permite que cambiar entre diferentes algoritmos sea relativamente sencillo, como solo cambiar el nombre del estimador que se invoca para el entrenamiento y evaluación.

En esta ocasión usaremos el archivo *02_SVM_SKLearnDigits.ipynb* con el Dataset Digits de SKLearn. En este caso el código suministrado esta incompleto y debe agregar los siguientes fragmentos, en los espacios indicados, para hacer lo funcionar. Este ejemplo ha sido adaptado del libro *Python For Programmers de Deitel & Deitel* [3].

En esta sección crearemos un clasificador de dígitos utilizando el algoritmo de *Support Vectors Machine* o SVM, mediante la implementación disponible en SKLearn. Se utiliza de igual manera el Dataset Digits, sin embargo en esta ocasión se usa la versión del dataset disponible en la librería SKLearn.

En este caso el descriptor consiste en un vector de 64 dimensiones, en el cual se toma el valor de cada pixel de manera lineal o consecutiva. Dicho de otra manera el valor de cada pixel se coloca en una dimensión del descriptor.

1. Con este fragmento creamos la instancia del estimador y definimos los parámetros básicos para la ejecución del mismo. En este caso el parámetro *kernel* indica la función que se utiliza en el modelo para estimar la separación o limites entre las distintas clases presentes en el dataset.

Celda 1

```
from sklearn.svm import SVC

svcclassifier = SVC(kernel='poly', degree=8, gamma='auto')
```

- Definición del estimador

2. Con el siguiente código se aplica un enfoque de Cross Validation para evaluar nuestro dataset con el el estimador SVM. Este enfoque permite crear k grupos de instancias y entrenar varios modelos utilizando los grupos, siempre dejando un grupo para hacer el test del modelo, ver Figura 3. Esto permite evaluar como de bien nos funcionan nuestros datos para entrenar un modelo.

Celda 2

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(estimator=svcclassifier,
                          X=digits.data, y=digits.target,
                          cv=kfold)
```

scores

- Aplicar Cross Validation con la función `cross_val_score`

- En este caso el dataset se importa del modulo `load_digits` de SciKitLearn el cual contiene un conjunto de dataset disponibles para ser utilizadas en el nuestros experimentos. Con el siguiente fragmento se llama al dataset a nuestro código.

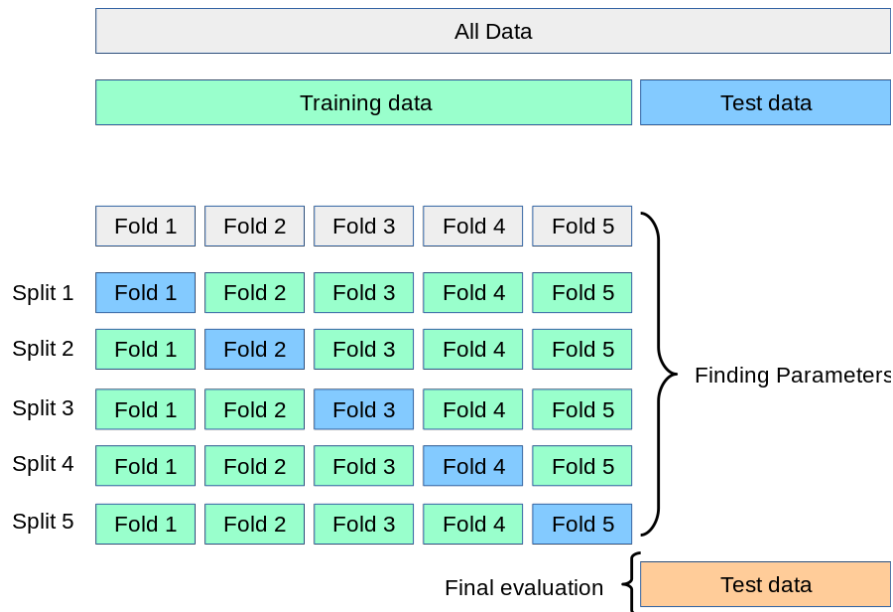


Figura 3: Esquema del funcionamiento de Cross Validation

```
digits = load_digits()
```

- El entrenamiento en un clasificador de SKLearn se realiza mediante la función `.fit()`, esta recibe como argumentos básicos la lista de descriptores del dataset para entrenamiento, así como también las etiquetas o valores reales (*ground truth* para cada imagen en el dataset. En el siguiente fragmento `X` hace referencia a una lista de vectores donde cada vector es una imagen. Mientras que `Y` hace referencia a la lista de categorías para cada animal en el dataset.

```
svclassifier.fit(X=X_train, y=y_train)
```

- El siguiente código permite hacer una predicción basado en el modelo. En este caso se debe enviar un argumento que debe contener una instancia o una lista de instancias/descriptores representados de la misma manera que las instancias de train, y de las cuales deseamos realizar una predicción.

```
predicted = svclassifier.predict(X=X_test)
```

- De igual manera podemos obtener el score del clasificador entrenado, enviado a este la lista de instancias/descriptores. En este caso se realiza evaluación mediante la secuencia de test del dataset.

```
print(f'{svclassifier.score(X_test, y_test):.2%}')
```

- Al final del notebook se presenta un código que se enfoca en el evaluar varios clasificadores utilizando Cross Validation (Ver Figura 3), de esta manera es posible encontrar cual podría ser la mejor opción para la creación de nuestro modelo. Se evalúan en este caso los clasificadores KNN, SVM y Naive bayes Gaussiano.

```
# Instancias de Clasificadores a evaluar
estimators = {
    'KNeighborsClassifier': KNeighborsClassifier(),
    'SVC': svcclassifier,
    'GaussianNB': GaussianNB()}

for estimator_name, estimator_object in estimators.items():
    kfold = KFold(n_splits=10, random_state=11, shuffle=True)
    scores = cross_val_score(estimator=estimator_object,
                             X=digits.data, y=digits.target, cv=kfold)
    print(f'{estimator_name:>20}: ' +
          f'mean accuracy={scores.mean():.2%}; ' +
          f'standard deviation={scores.std():.2%}')
```

Entregables para la Sección

1. Defina el kernel del modelo a un tipo rbf, ejecute el sistema e indique los resultados.
[Información sobre RBF Kernel.](#)
2. Teniendo en cuenta la matriz de confusión generada con el programa original, indique que podemos deducir de esta, si nos enfocamos en el número 1.
3. Aplique un k-fold con $k = 25$ y reporte cual es la exactitud promedio obtenida por el modelo con RBF.

3. Random Forest

En esta sección se utilizará un dataset que debe estar disponible en su computadora para entrenar un modelo de clasificación mediante el algoritmo Random Forest. Como primer paso se procede a presentar el dataset a utilizar el cual debe estar descargado en una ubicación de su computadora.

3.1. Dataset para el laboratorio

Utilizaremos un dataset que emplea las imágenes de las categorías “Cat”, “Dog” y “Parrot” del dataset Imagenet y recopiladas de Google Images. Para ello, el dataset ya viene separado en las secuencias de entrenamiento(*train*), *test* y validación(*validation*), los cuales se utilizan para entrenar, verificar y evaluar el modelo, respectivamente.

Una vez descargue el dataset (Disponible en [Pets_Dataset](#)), debe verificar que mantenga la estructura requerida, la cual se aprecia en las siguientes imágenes. La carpeta raíz recibe el nombre de “pets_dataset”. El cual debe tener la siguiente estructura una vez que se abre su carpeta contenedora (Ver Figura 4). Disponible también en Google Drive [Pets_Dataset](#).

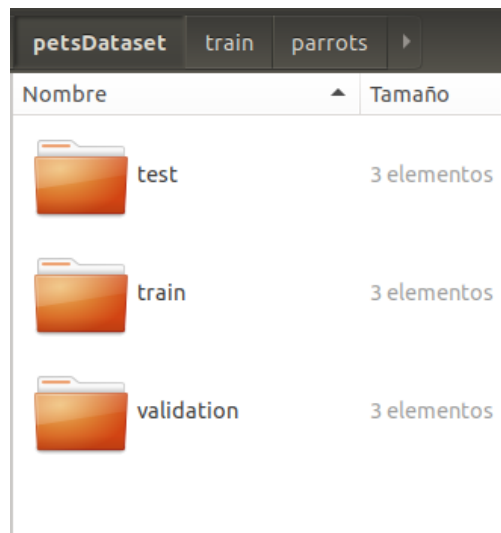
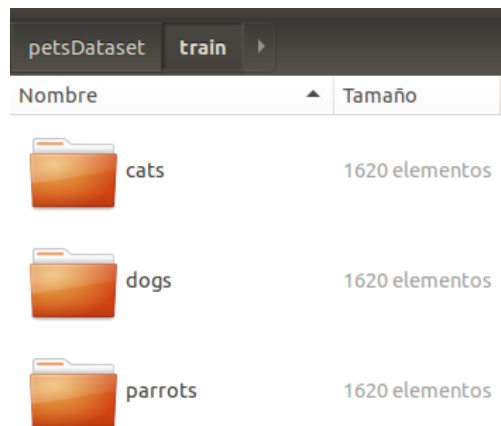
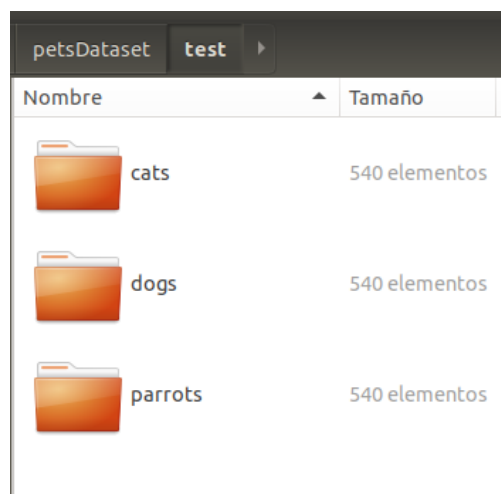


Figura 4: Carpeta Raíz del Dataset.

Dentro de cada una de estas carpetas se encuentran con conjunto de imágenes separadas en 3 clases/categorías. Siendo estas gatos, perros y loros. El interior de cada una de estas carpetas se debe ver como las Figuras 5, 6, 7, para los conjuntos de Entrenamiento, Test y Validación, respectivamente:



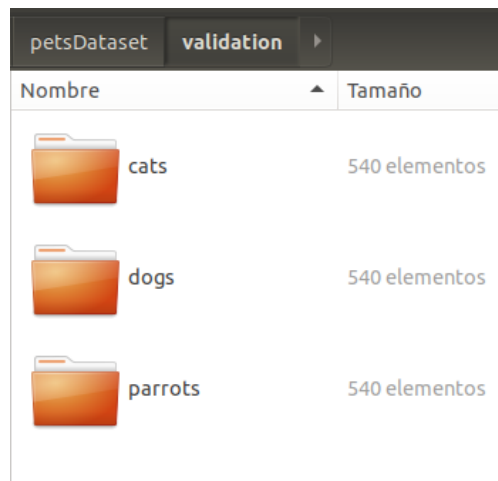
Nombre	Tamaño
cats	1620 elementos
dogs	1620 elementos
parrots	1620 elementos

Figura 5: Carpeta *Train* del Dataset

Nombre	Tamaño
cats	540 elementos
dogs	540 elementos
parrots	540 elementos

Figura 6: Carpeta *Test* del Dataset

Observe la distribución de imágenes de cada categoría en las carpetas del dataset. Es importante mantener en la medida de lo posible las proporciones recomendadas de 60 %/20 %/20 % para las secuencias de *train/test/validation* respectivamente. En este laboratorio se cuenta con un dataset que tiene la misma cantidad de imágenes para cada categorías, por lo cual es posible



Nombre	Tamaño
cats	540 elementos
dogs	540 elementos
parrots	540 elementos

Figura 7: Carpeta Validation del Dataset

una división tan precisa y balanceada de los datos, sin embargo, esto no es siempre posible y suele suceder que se tiene más información de una clase que de otra, por lo cual, en ese caso las proporciones se deben cuidar también, pero basándonos en la cantidad de elementos de clase.

3.2. Entrenamiento del Modelo

Para entrenar nuestro modelo de clasificación de mascotas usaremos el enfoque de Random Forest. Se utilizará el archivo `03_RF_Hog_Pets.ipynb`. En este caso se usará solamente la secuencia de *Train* y *Test* del dataset presentado.

- Como primer paso de esta sección debemos configurar la ruta en la cual se encuentra nuestro dataset en la computadora, para ello debe descargar el dataset e indicar en la siguiente línea, cual es la ubicación donde descomprimirá el dataset en su computadora. En caso de trabajar con Google Colab, debe subir su dataset al drive y hacer la conexión de su notebook con el Google Drive.

```
#Definición de los directorios del dataset
base_dir = r'C:\Users\jockr\Desarrollo\petsDataset'
```

- Las líneas de código siguientes se enfocan en crear las rutas a las carpetas respectivas, para permitir que se construya una lista con la ruta completa de cada imagen en el dataset. En el siguiente fragmento, se construye la lista de los elementos dentro de una carpeta y seguidamente, se añade a cada elemento, la ruta del directorio base de tal manera que se cuenta con la ruta completa a la imagen.

```
# Lista de Nombres de Archivos de Train
train_cat_fnames = os.listdir(train_cats_dir )
```

```
# Lista de Rutas de archivos de train
train_cat_fnames = list(map(lambda x: os.path.join(train_cats_dir, x) , train_cat_fnames))
```

- El siguiente código crea una lista paralela a la lista de todos los archivos, en el cual se almacena la categoría a la cual pertenece cada imagen en el dataset, se crea una lista de labels tanto para *train* como para *test*. En este caso cada categoría se le asigna un valor numérico siendo 1, 2 o 3, para indicar gato, perro y loro, respectivamente.

```
# Prepara las etiquetas de cada instancia de train
train_labels =[ 0 for _ in train_cat_fnames ]
train_labels.extend([ 1 for _ in train_dog_fnames ])
train_labels.extend([ 2 for _ in train_parrot_fnames ])
```

- Las siguientes funciones nos permiten generar la matriz de confusión de nuestro modelo, ya sea usando el modelo directamente o solo enviando las predicciones.

```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

def generarMC(model, X, y, categorias, normal=None ):
    fig=ConfusionMatrixDisplay.from_estimator(model, X, y,
                                              display_labels=categorias,
                                              cmap="Greens",
                                              normalize=normal)

    fig.figure_.suptitle("Confusion Matrix")
    plt.show()

def generarMCPredictions(y_real, y_pred, categorias ):
    fig=ConfusionMatrixDisplay.from_estimator(y_real, y_pred,
                                              display_labels=categorias,
                                              cmap="Greens")

    fig.figure_.suptitle("Confusion Matrix")
    plt.show()
```

- La función `get_hog(list_fnames)` recibe una lista de rutas de archivos y calcula el descriptor global HOG [2] para cada uno, al final retorna una lista con los descriptores de cada imagen en la lista original. El descriptor HOG recibe ciertos parámetros para realizar su cálculo. El tamaño final de este descriptor (cantidad de dimensiones) depende del tamaño de la imagen y de ciertos parámetros como tamaño de bloques y celdas que se definen para el computo del descriptor. Para asegurar que el descriptor de cada imagen tenga el mismo tamaño se define un tamaño de imagen y se aplica un *resize* a cada imagen antes de ser procesada.

```
# Definir Resolucion de las imagenes a usar
img_size =(380,380)
```

```
# Definir los parámetros del descriptor HOG
win_size = img_size # Tamaño de la ventana de detección
block_size = (40, 40) # Tamaño del bloque
block_stride = (20, 20) # Desplazamiento del bloque
cell_size = (20, 20) # Tamaño de la celda
nbins = 9 # Número de bins del histograma

# Crear un descriptor HOG con parámetros personalizados
hog = cv2.HOGDescriptor(win_size, block_size, block_stride, cell_size, nbins)
```

- El siguiente fragmento utiliza la función descrita anteriormente para obtener el descriptor para cada imagen en la secuencia de *Train* y *Test*.

```
train_hog = get_hog(train_fnames)
test_hog = get_hog(test_fnames)
```

- Como siguiente punto se procede a realizar el entrenamiento de nuestro modelo con Random Forest.

```
from sklearn.ensemble import RandomForestClassifier

#Definición del Modelo
clfRF = RandomForestClassifier(n_jobs=6,
                              random_state=0,
                              n_estimators=1000,
                              )

#fit/train -> Se hace el entrenando del modelo
clfRF.fit(train_hog, np.array(train_labels))
```

- Como último punto se evalúa nuestro modelo utilizando la secuencia de *test* y sus respectivos valores reales.

```
# Precisión lograda por el clasificador
print(f'{clfRF.score(test_hog, np.array(test_labels)):.2%}')
```

- En la parte final del archivo se presenta una implementación usando la librería Gradio para crear una interfaz sencilla para probar nuestro clasificador.
- Este código define una interface de Gradio que tiene como entrada un campo de imagen, en este caso nuestro programa necesita la ruta de la imagen, por lo cual se define que del objeto imagen se necesita el path mediante el parámetro `type="filepath"`. De igual manera se define la salida que debe mostrar la interfaz y se especifica que es de tipo texto, por lo cual nuestra interfaz mostrar un cuadro de texto para presentar la salida. El parámetro `fn` recibe el nombre de la función que se ejecuta cuando se presiona el boton *submit* en la interface, los parámetros que recibe y retorna esta función deben coincidir con los *inputs* y *outputs* declarados en para la interfaz.

```
import gradio as gr

gr.Interface(
    fn = clasificar, # funcion
    inputs = gr.Image(type="filepath"), # tipo de entrada
    outputs = "textbox"
).launch()
```

- La función `def clasificar(path)` define el método que será invocado desde la interfaz de Gradio cuando se presione el botón *Submit*. Esta función recibe una ruta de la imagen y devuelve un string con el resultado de la clasificación. En este caso la ruta de la imagen se envía a la función `get_hog` para obtener el descriptor de la imagen. Este descriptor se envía a nuestro clasificador previamente entrenado y la respuesta de este se utiliza para obtener la categoría asignada a la imagen enviada.

```
def clasificar(path):
    global clfRF
    hogDesc = get_hog([path])
    labels = ['cat', 'dog', 'parrot']
    res = clfRF.predict(hogDesc.reshape(1, -1))
    resProb = clfRF.predict_proba(hogDesc.reshape(1, -1))
    labelResultado = f'En la imagen aparece un {labels[res[0]]},
    con una probabilidad de { resProb[0,res[0]]:.2%}'
    return labelResultado
```

- En el método anterior la línea `resProb = clfRF.predict_proba(hogDesc.reshape(1, -1))` se utiliza para obtener el valor de probabilidad que ha calculado nuestro clasificador para la pertenencia de la imagen a cada una de las categorías disponibles en el dataset. El método `predict` que se usa más frecuentemente nos da como resultado el índice de la categoría con la probabilidad mayor.

Entregables para la Sección

1. Prepare los datos de la secuencia de validación disponibles en el dataset
2. Pruebe su modelo (Calcular Score) de Random Forest utilizando la secuencia de Validación y presente la Matriz de Confusión Normalizada obtenida para esta secuencia.
3. Pruebe su modelo (Calcular Score) de Random Forest utilizando la secuencia de Train y presente la Matriz de Confusión Normalizada obtenida para esta secuencia.
4. Presente una captura de pantalla de una prueba de este clasificador utilizando Gradio.

4. Redes Neuronales usando SKLearn

En este caso se utilizarán **Perceptrones(MLPs)**, para entrenar un modelo mediante SKLearn, estos son también conocidos como Redes Neuronales. Para este modelo se utilizará el dataset `digits` de la sección 1 y el descriptor global HOG [2], para representar las imágenes. Por lo cual, las primeras funciones del código, ya han sido explicadas anteriormente y solo se detallan los nuevos códigos.

En esta sección el archivo de código se llama: `04_MLP_Hog.ipynb`

- Como primer punto, luego de generados los descriptores con HOG, se procede a realizar la partición del dataset en un conjunto de entrenamiento y de prueba, en este caso se utiliza al función `train_test_split()` de SKLearn. Esta función recibe la lista de descriptores, las etiquetas correspondientes y el tamaño que se desea conserve la secuencia de Test, en este caso definido como el 30 %.

```
# At this point we split the data into training and testing (70% for training):
hog_descriptors_train, hog_descriptors_test, labels_train, labels_test =
    train_test_split(
        hog_descriptors,
        labels, random_state=1,
        test_size=0.3)
```

- La función devuelve 4 listas en las cuales están los descriptores de cada instancia o imagen y su respectiva etiqueta, separadas en *Train* y *Test*.
- Una vez separadas las instancias, se procede a crear nuestra instancia de estimador/clasificador en este caso un MLP.

```
clf = MLPClassifier(hidden_layer_sizes=(256,128,64,32),activation="relu",random_state=25)
```

- Esta línea de código crea la instancia del MLP, el primer parámetro, es una lista de valores enteros que define el tamaño o cantidad de LTUs que tendrá cada capa oculta de la red o perceptron. La cantidad de capas ocultas se define por la cantidad o largo de esta lista. En nuestro caso definimos 4 capas con tamaños (256, 128, 64, 32) respectivamente.
- El tamaño de la capa de entrada y la capa de salida, se definen de manera automática por SKLearn, dependiendo de la cantidad de dimensiones de los descriptores y la cantidad de clases que tiene el dataset.
- Para el entrenamiento del modelo se procede a utilizar el método `fit()`, este recibe el dataset que se utilizará para entrenar el modelo y las etiquetas de este dataset.

```
clf.fit(hog_descriptors_train, labels_train)
```

- Luego de entrenado el modelo se puede utilizar el método `score()`, para calcular el accuracy del modelo. Este método recibe el dataset de test (en este caso) y las etiquetas de cada instancia en el dataset.

```
print(clf.score(hog_descriptors_test, labels_test))
```

- Como último punto se procede a generar la matriz del confusión para el modelo, en este caso generada a partir del modelo. La función recibe el modelo, el dataset y las etiquetas para crear la matriz del modelo indicado con los datos enviados.

```
ConfusionMatrixDisplay.from_estimator(
    clf,
    hog_descriptors_test,
    labels_test,
    normalize='true',
    cmap='Greens')
plt.title("Matriz de Confusión para Test")
plt.show()
```

- En esta caso se utiliza el parámetro `normalize`, para que el resultado de cada categoría se presente con un valor entre 0 y 1, para realizar una interpretación más sencilla.

Entregables para la Sección

1. Realice 2 modificaciones las capas definidas del MLP y genere para cada modificación la matriz de confusión de *Train* y *Test*. Puede modificar tamaños de capa y cantidad de capas o ambas. Puede indagar otros parámetros de esta función y mortificarlos para evaluar su efecto en el resultado final.
2. Coloque adecuadamente el titulo a cada matriz y presente la matriz en su informe de laboratorio, indique para cada una la cantidad de capas ocultas y el tamaño de cada una, así como también el accuracy obtenido.
3. Utilizando el código y secuencias de *train* y *test* de la Sección 3 reemplace el clasificador de Random Forest por un MLP y presente la matriz de confusión para este nuevo modelo entrenado.
4. Presente una captura de pantalla de una prueba de este clasificador utilizando Gradio.

Referencias

- [1] A. Fernández Villán, *Mastering OpenCV 4 with Python: a practical guide covering topics from image processing, augmented reality to deep learning with OpenCV 4 and Python 3.7. Mastering Open Source Computer Vision four with Python*. Birmingham: Packt Publishing, 2019. [Online]. Available: <https://cds.cern.ch/record/2674578>
- [2] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, pp. 886–893 vol. 1.
- [3] P. Deitel and H. Deitel, *Python for Programmers*. Pearson Education, 2019. [Online]. Available: <https://books.google.com.pa/books?id=LauMDwAAQBAJ>