Importamos las librerias que vamos a utilizar:

```
import numpy as np
import pandas as pd
import os
import cv2 as cv
import matplotlib.pyplot as plt
import seaborn as sns
```

numpy y pandas son bibliotecas para manipulación de datos y cálculos numéricos.

os se utiliza para operaciones relacionadas con el sistema de archivos.

cv2 es una parte de la biblioteca OpenCV, que se usa comúnmente para procesamiento de imágenes y visión por computadora.

matplotlib.pyplot se utiliza para trazar gráficos y visualizar datos.

seaborn es una biblioteca de visualización de datos que funciona bien con Pandas y Matplotlib.

Agregamos la ruta a cada una de nuestras carpetas.

```
all_0 = "./C-NMC_Leukemia/training_data/fold_0/all"
all_1 = "./C-NMC_Leukemia/training_data/fold_1/all"
all_2 = "./C-NMC_Leukemia/training_data/fold_2/all"

hem_0 = "./C-NMC_Leukemia/training_data/fold_0/hem"
hem_1 = "./C-NMC_Leukemia/training_data/fold_1/hem"
hem_2 = "./C-NMC_Leukemia/training_data/fold_2/hem"
```

En esta sección, se definen rutas de directorio para seis carpetas diferentes. Estas rutas contienen nuestros datos de entrenamiento para la tarea de clasificación llamada "C-NMC_Leukemia". Los datos se dividen en tres carpetas "**all**" y tres carpetas "**hem**" correspondientes a tres particiones diferentes llamadas "fold_0", "fold_1" y "fold_2".

Recordemos que hem representa a células sin leucemia y, por otro lado, ALL representa células con Leucemia linfoblástica aguda .
Esta celda, que almacenan las rutas a las carpetas en texto, la utilizaremos más adelante en

el código para poder acceder a las imágenes, haremos uso de la librería OS para poder acceder a los archivos de la computadora.

```
def get_path_image(folder):
    image_paths = []
    image_fnames = os.listdir(folder)
    for img_id in range(len(image_fnames)):
        img = os.path.join(folder,image_fnames[img_id])
        image_paths.append(img)

    return image_paths
```

La función *get_path_image():* toma una ruta de directorio folder como entrada y devuelve una lista de rutas de archivo. Aquí se detallan los pasos que sigue la función:

image_paths es una lista vacía que se utilizará para almacenar las rutas de imagen.

image_fnames es una lista de nombres de archivo en el directorio folder obtenidos usando os.listdir(folder).

Luego, se recorre la lista de nombres de archivo en un bucle for. Para cada nombre de archivo, se crea una ruta completa concatenando el nombre del archivo con la ruta del directorio folder utilizando os.path.join(). Luego, la ruta completa se agrega a la lista image_paths.

Finalmente, la función devuelve la lista de rutas de imagen.

Hasta ahora nuestro código importa bibliotecas necesarias, define rutas de directorio para cargar imágenes y proporciona una función get_path_image para obtener rutas de imágenes en un directorio específico. Este código es el inicio del proyecto de procesamiento de imágenes y visión por computadora relacionados con la clasificación de células sanguíneas en la tarea de "C-NMC_Leukemia".

Almacenemos todas las rutas de nuestras imágenes en una lista:

```
img_data = []

for i in [all_0,all_1,all_2,hem_0,hem_1,hem_2]:
    paths = get_path_image(i)
    img_data.extend(paths)
print(len(img_data))
```

img_data = []: Se inicializa una lista vacía llamada img_data que se utilizará para almacenar las rutas completas de las imágenes.

for i in [all_0, all_1, all_2, hem_0, hem_1, hem_2]: Inicia un bucle for que recorre una lista de rutas de directorios. Esta lista contiene las rutas a las carpetas "all" y "hem" correspondientes a diferentes particiones de datos. La intención aquí es recopilar rutas de imágenes de ambas clases ("all" y "hem").

paths = get_path_image(i): Para cada directorio en el bucle for, se llama a la función get_path_image(i) para obtener una lista de rutas completas de imágenes en ese directorio. La variable paths almacena estas rutas.

img_data.extend(paths): Se utiliza el método extend() de la lista img_data para agregar las rutas de imágenes obtenidas en el paso anterior a la lista img_data. Esto se hace para acumular todas las rutas en una sola lista en lugar de tener múltiples listas separadas. print(len(img_data)): Al final del bucle for, se imprime la longitud de la lista img_data, que corresponde al número total de rutas de imágenes recopiladas de todas las carpetas "all" y "hem". Esto proporciona una idea de cuántas imágenes hay en total en las carpetas (10660) y serviría para verificar si se han recopilado correctamente todas las rutas de las imágenes.

Empezamos con el preprocesamiento de nuestros datos/imágenes:

```
data = {"img_data":img_data,
        "labels":[np.nan for x in range(len(img_data))]}

data = pd.DataFrame(data)

data["labels"][0:7272] = 1 # ALL
data["labels"][7272:10661] = 0 # HEM

data["labels"] = data["labels"].astype("int64")

image = cv.imread(data["img_data"][1000])
plt.imshow(image)
plt.title("Sample image before cropping")
plt.show()
```
+

data = {"img_data": img_data, "labels": [np.nan for x in range(len(img_data))]}

Aquí, se crea un diccionario llamado data. Este diccionario tiene dos claves: "img_data" y "labels".

"img_data" contiene la lista de rutas de imágenes img_data.

"labels" se inicializa con una lista de np.nan (valores NaN) del mismo tamaño que la lista de rutas de imágenes. Esto se hace para inicializar todas las etiquetas con valores no definidos.

data = pd.DataFrame(data)

Se convierte el diccionario data en un DataFrame de Pandas. Esto crea una tabla donde las columnas son "img_data" y "labels", y las filas contienen las rutas de imágenes y las etiquetas respectivas.

data["labels"][0:7272] = 1 # ALL

En esta línea, se asigna el valor 1 a las filas desde la posición 0 hasta la 7271 en la columna "labels" del DataFrame. Estas son las etiquetas correspondientes a la clase "ALL".

data["labels"][7272:10661] = 0 # HEM

Aquí, se asigna el valor 0 a las filas desde la posición 7272 hasta la 10660 en la columna "labels" del DataFrame. Estas son las etiquetas correspondientes a la clase "HEM".

data["labels"] = data["labels"].astype("int64")

Se convierten los valores en la columna "labels" del DataFrame en enteros de 64 bits. Esto es importante para asegurarse de que las etiquetas sean de tipo entero para tareas de clasificación.

image = cv.imread(data["img_data"][1000])

Se utiliza la biblioteca OpenCV para cargar una imagen de la ruta especificada en la fila 1000 de la columna "img_data" del DataFrame data.

plt.imshow(image)

Se muestra la imagen cargada utilizando Matplotlib. Esto permite visualizar la imagen en una ventana gráfica.

plt.title("Sample image before cropping")

Se agrega un título a la imagen que se muestra en la ventana gráfica.

plt.show()

Finalmente, se muestra la ventana gráfica con la imagen en ella.

Ahora modificaremos nuestras imágenes para que nuestro modelo logre un mayor rendimiento.

```
img_list = []
for i in range(len(img_data)):
    image = cv.imread(data["img_data"][i])
    gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    thresh = cv.threshold(gray, 0, 255, cv.THRESH_BINARY_INV + cv.THRESH_OTSU)[1]

    result = cv.bitwise_and(image, image, mask=thresh)
    result[thresh==0] = [255,255,255]
    (x, y, z_) = np.where(result > 0)
    mnx = (np.min(x))
    mxx = (np.max(x))
    mny = (np.min(y))
    mxy = (np.max(y))
    crop_img = image[mnx:mxx,mny:mxy,:]
    crop_img_r = cv.resize(crop_img, (224,224))
    img_list.append(crop_img_r)
```

img_list = []: Se inicializa una lista vacía llamada img_list que se utilizará para almacenar las imágenes procesadas.

for i in range(len(img_data)):: Se inicia un bucle for que recorre todas las rutas de imágenes almacenadas en img_data.

image = cv.imread(data["img_data"][i]): Se utiliza OpenCV para cargar una imagen desde la ruta especificada en data["img_data"][i]. La imagen se almacena en la variable image.

gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY): La imagen se convierte a escala de grises utilizando cv.cvtColor(). Esto es común en procesamiento de imágenes para reducir la complejidad y, en algunos casos, mejorar el rendimiento en tareas de clasificación.

thresh = cv.threshold(gray, 0, 255, cv.THRESH_BINARY_INV + cv.THRESH_OTSU)[1]: Se realiza una operación de umbralización en la imagen en escala de grises para separar las áreas de interés de las áreas de fondo. El método cv.THRESH_BINARY_INV + cv.THRESH_OTSU se utiliza para calcular el umbral automáticamente utilizando el algoritmo de Otsu. El resultado se almacena en la variable thresh.

result = cv.bitwise_and(image, image, mask=thresh):

Se utiliza la máscara thresh para aplicar una operación bitwise_and(Esta función es parte de las operaciones de procesamiento de imágenes y se utiliza para diversas tareas, como la segmentación de objetos, la eliminación de fondos y la combinación de imágenes.

La operación bitwise_and se realiza tomando los valores de píxeles de dos imágenes (o una imagen y una máscara) y aplicando la operación lógica "y" a nivel de bits a estos valores.)
a la imagen original. Esto preserva las áreas de interés y elimina el fondo. El resultado se almacena en la variable result.

result[thresh==0] = [255,255,255]: Se asigna el color blanco (255, 255, 255 en el espacio de color BGR) a todas las áreas donde la máscara thresh tiene un valor de 0, es decir, el fondo. Esto elimina el fondo y lo reemplaza por blanco.

(x, y, z_) = np.where(result > 0): Se utiliza NumPy (np.where) para encontrar las coordenadas de los píxeles que tienen un valor mayor que 0 en la imagen procesada result.

mnx = (np.min(x)), mxx = (np.max(x)), mny = (np.min(y)), mxy = (np.max(y)): Se calculan los valores mínimos y máximos de las coordenadas x e y de los píxeles de interés. Estos valores se utilizan para recortar la imagen.

crop_img = image[mnx:mxx, mny:mxy, :]: Se recorta la imagen original utilizando los valores mínimos y máximos de coordenadas obtenidos anteriormente. Esto recorta la imagen para mantener solo las regiones de interés.

crop_img_r = cv.resize(crop_img, (224, 224)): La imagen recortada se redimensiona a un tamaño específico de 224x224 píxeles. Este tamaño puede ser necesario para adaptar las imágenes a un modelo de aprendizaje automático que requiera un tamaño de entrada específico.

img_list.append(crop_img_r): La imagen procesada y redimensionada se agrega a la lista img_list. Esto se hace para acumular todas las imágenes procesadas.

Esta celda lo que hace es:
realiza una serie de operaciones de procesamiento de imágenes, incluida la conversión a escala de grises, umbralización, eliminación del fondo, recorte y redimensionamiento de las imágenes antes de agregarlas a la lista img_list. Estas operaciones son comunes en el preprocesamiento de imágenes antes de utilizarlas en tareas de clasificación o análisis.

Hasta acá llega el preprocesamiento de nuestros datos.

Empecemos a importar nuevas librerías y empecemos a preparar nuestros datos para que sean entrenados y testeados:

```
from tensorflow.keras.applications import ResNet50, ResNet101
from keras.applications.vgg19 import VGG19
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Model
from tensorflow.keras.applications.resnet50 import preprocess_input
```

ResNet50 y ResNet101 de tensorflow.keras.applications:

ResNet (Redes Neuronales Residuales) es una arquitectura de red neuronal profunda que se ha destacado en tareas de visión por computadora y clasificación de imágenes. Los modelos ResNet50 y ResNet101 son versiones preentrenadas de ResNet con 50 y 101 capas, respectivamente.

Estos modelos se utilizan para la extracción de características y clasificación de imágenes. Cada uno de ellos es capaz de reconocer y extraer características de una amplia variedad de objetos y patrones en imágenes. Se entrenaron en un gran conjunto de datos y, por lo tanto, pueden ser útiles para tareas de clasificación de imágenes.

Los modelos ResNet utilizan conexiones residuales para abordar el problema del desvanecimiento del gradiente en redes profundas. Esto permite entrenar redes neuronales más profundas de manera efectiva.

VGG19 de keras.applications:

VGG (Visual Geometry Group) es otra arquitectura de red neuronal profunda que es conocida por su simplicidad y efectividad. VGG19 es un modelo preentrenado de VGG con 19 capas.Al igual que los modelos ResNet, VGG19 se utiliza para la clasificación de imágenes y la extracción de características. Puede reconocer una amplia variedad de objetos y patrones en imágenes.

La arquitectura VGG se caracteriza por tener capas convolucionales de 3x3 con muchas capas apiladas, lo que la hace profunda. Esto permite aprender representaciones de características más complejas en imágenes.

image de tensorflow.keras.preprocessing:

image es un módulo que proporciona herramientas para la carga, preprocesamiento y aumento de imágenes. Se utiliza para preparar imágenes para la alimentación a modelos de aprendizaje automático. Incluye funciones para cargar imágenes desde archivos, cambiar el tamaño de las imágenes y aplicar transformaciones.

Model de tensorflow.keras.models:

Model es una clase que se utiliza para construir y definir modelos de aprendizaje automático en TensorFlow y Keras. Permite definir modelos de forma personalizada al especificar las capas y las conexiones entre ellas.

preprocess_input de tensorflow.keras.applications.resnet50:

preprocess_input es una función que se utiliza para preprocesar imágenes antes de alimentarlas a modelos preentrenados como ResNet50. Aplica transformaciones específicas, como la resta de la media, al conjunto de datos de imágenes para que coincidan con la forma en que se entrenó el modelo.

```
def feature_extract(model):
    if model == "VGG19": model = VGG19(weights='imagenet',include_top=False, pooling="avg")
    elif model == "ResNet50": model = ResNet50(weights='imagenet',include_top=False,pooling="avg")
    elif model == "ResNet101": model = ResNet101(weights='imagenet',include_top=False,pooling="avg")
    return model
```

Este código define una función llamada feature_extract que toma un argumento model y devuelve un modelo de redes neuronales preentrenado específico según el valor de model. La función se encarga de cargar y configurar el modelo seleccionado para la extracción de características. Aquí está una explicación línea por línea:

def feature_extract(model): Se define una función llamada feature_extract que acepta un argumento model.

if model == "VGG19": model = VGG19(weights='imagenet', include_top=False, pooling="avg"): Si el valor de model es igual a la cadena "VGG19", la función crea un modelo VGG19 preentrenado utilizando la función VGG19 de Keras. Se especifican varios argumentos para configurar el modelo:

weights='imagenet': Carga los pesos preentrenados en el modelo.

include_top=False: No incluye la capa superior del modelo, que generalmente es la capa de salida de clasificación.

pooling="avg": Configura el tipo de operación de pooling, que en este caso es un promedio global.

elif model == "ResNet50": model = ResNet50(weights='imagenet', include_top=False, pooling="avg"): Si el valor de model es igual a la cadena "ResNet50", la función crea un modelo ResNet50 preentrenado. Los argumentos son similares a los del caso anterior.

elif model == "ResNet101": model = ResNet101(weights='imagenet', include_top=False, pooling="avg"): Si el valor de model es igual a la cadena "ResNet101", la función crea un modelo ResNet101 preentrenado, con argumentos similares a los anteriores.

return model: La función devuelve el modelo configurado. Dependiendo del valor de model proporcionado como argumento, la función retornará un modelo VGG19, ResNet50 o ResNet101 preentrenado, configurado según las especificaciones mencionadas.

En resumen, esta función feature_extract permite seleccionar y configurar un modelo de redes neuronales preentrenado (VGG19, ResNet50 o ResNet101) según el valor de model proporcionado como argumento. El modelo seleccionado se configura para la extracción de características y se devuelve para su posterior uso en la extracción de características de imágenes.

```
] model = feature_extract("ResNet50") # or "VGG19", "ResNet101"

features_list = []
for i in range(len(img_list)):

    image = img_list[i].reshape(-1, 224, 224, 3)
    image = preprocess_input(image)

    """
    # Reshaping when VGG19 model is selected
    features = model.predict(image).reshape(512,)
    """

    #Reshaping  when ResNet50 or ResNet101 model is selected
    features = model.predict(image).reshape(2048,)

    features_list.append(features)
```

Esta parte del código utiliza la función feature_extract para seleccionar un modelo preentrenado (en este caso, "ResNet50") y luego utiliza ese modelo para extraer características de las imágenes en img_list. Aquí está la explicación línea por línea:

model = feature_extract("ResNet50"): Se llama a la función feature_extract con el argumento "ResNet50", lo que significa que se selecciona y configura el modelo ResNet50

preentrenado. El modelo se almacena en la variable model y se utilizará para la extracción de características.

features_list = []: Se inicializa una lista vacía llamada features_list que se utilizará para almacenar las características extraídas de las imágenes.

for i in range(len(img_list)):: Se inicia un bucle for que recorre todas las imágenes en la lista img_list.

image = img_list[i].reshape(-1, 224, 224, 3): Se selecciona una imagen de la lista img_list en la posición i y se realiza un cambio en su forma (reshape) para que tenga dimensiones de (-1, 224, 224, 3). Esto es necesario para que la imagen sea compatible con el modelo seleccionado. Las dimensiones (224, 224, 3) se refieren a un tamaño de imagen de 224x224 píxeles con 3 canales de color (RGB).

image = ***preprocess_input(image)***: La imagen se preprocesa utilizando la función preprocess_input. Esta función aplica transformaciones específicas al conjunto de datos de imágenes para que coincidan con la forma en que se entrenó el modelo. Esto asegura que las imágenes estén en el formato correcto antes de ser procesadas por el modelo.

features = model.predict(image).reshape(2048,): El modelo seleccionado (model) se utiliza para predecir características en la imagen preprocesada (image) utilizando el método predict. Dependiendo del modelo seleccionado (en este caso, ResNet50), las características se reformatean para tener una forma específica, que es de 2048 en este caso. Las características resultantes se almacenan en la variable features.

features_list.append(features): Las características extraídas de la imagen se agregan a la lista features_list. Esto se hace para acumular las características extraídas de todas las imágenes en img_list.

Que es preprocess_input y para que lo usamos:

```
image = preprocess_input(image)
```

En ese caso, La línea de código que importamos al comienzo: *from tensorflow.keras.applications.resnet50 import preprocess_input* importa una función llamada preprocess_input desde el módulo resnet50 del paquete

applications en TensorFlow y Keras. Esta función se utiliza para preprocesar imágenes antes de alimentarlas a modelos de aprendizaje profundo, en particular, modelos como ResNet50. Aquí está su utilidad:

Preprocesamiento de Imágenes: La función preprocess_input se utiliza para aplicar transformaciones específicas a las imágenes antes de pasarlas a un modelo preentrenado, en este caso, ResNet50. Estas transformaciones incluyen la sustracción de la media y la adaptación de las imágenes al formato y la escala que se utilizaron durante el entrenamiento del modelo.

Compatibilidad con el Modelo: Los modelos preentrenados, como ResNet50, generalmente requieren que las imágenes de entrada cumplan con ciertas expectativas en términos de formato y preprocesamiento. La función preprocess_input asegura que las imágenes estén en el formato adecuado para ser compatibles con el modelo seleccionado.

Mejora del Rendimiento del Modelo: El preprocesamiento adecuado de imágenes es importante para el rendimiento del modelo. Al aplicar las mismas transformaciones que se utilizaron durante el entrenamiento, se garantiza que el modelo pueda interpretar las imágenes de manera coherente y realizar predicciones precisas.

Por lo tanto, la importación de preprocess_input desde tensorflow.keras.applications.resnet50 es esencial para garantizar que las imágenes se preprocesen correctamente antes de ser alimentadas a un modelo ResNet50 preentrenado, lo que contribuye a un funcionamiento preciso y consistente del modelo en tareas de visión por computadora.

Eso vendría a ser que es la función Preprocess_input(): Ahora se procede a describir que utilidad tiene en el código de clasificación de células con y sin leucemia:

la función preprocess_input se usa de la siguiente manera:

Seleccionas una imagen de la lista img_list en la posición i: image = img_list[i].

Luego, redimensionas la imagen utilizando image = img_list[i].reshape(-1, 224, 224, 3). Esta línea cambia la forma de la imagen a (-1, 224, 224, 3) para que coincida con las

dimensiones necesarias para la entrada del modelo. Sin embargo, antes de la redimensión, se debe preprocesar la imagen.

Aquí es donde entra en juego la función preprocess_input. La siguiente línea image = preprocess_input(image) aplica el preprocesamiento adecuado a la imagen. Esta función realiza las transformaciones necesarias en la imagen para que esté en el formato correcto y tenga la escala adecuada antes de alimentarla al modelo preentrenado. El modelo ResNet50 espera que las imágenes estén preprocesadas de esta manera.

Después del preprocesamiento, se pasa la imagen preprocesada al modelo para extraer características, dependiendo del modelo seleccionado. En este caso, el modelo es ResNet50, por lo que se realiza la predicción y se extraen características.

La función preprocess_input es crucial para garantizar que las imágenes se preprocesen de la manera esperada por el modelo. Esto es esencial para obtener resultados precisos y coherentes cuando se utilizan modelos preentrenados como ResNet50 en tareas de visión por computadora. La función preprocess_input realiza las transformaciones necesarias, como la sustracción de la media y la adaptación al rango y el formato adecuados, para que las imágenes sean compatibles con el modelo y produzcan resultados confiables. Debemos de prestar mucha atención a la lista: features_list = []
Al comienzo se inicializa vacia, pero: La lista features_list almacena características extraídas de las imágenes. Las características extraídas dependen del modelo preentrenado utilizado para la extracción. En este caso, el modelo seleccionado es ResNet50, y el valor reshape(2048,) sugiere que se están extrayendo 2048 características para cada imagen

Las características extraídas por modelos de redes neuronales convolucionales (CNN) como ResNet50 suelen ser **vectores numéricos**. Estos vectores representan representaciones de alto nivel de las imágenes en función de las características aprendidas por el modelo durante su entrenamiento en un gran conjunto de datos de imágenes. Las características pueden representar *patrones*, *texturas*, *formas* y otros atributos importantes presentes en las imágenes.

La coma en la línea reshape(2048,) se utiliza para especificar la forma deseada de la matriz resultante después de la operación de cambio de forma. En este caso, significa que se

espera que el resultado sea un vector unidimensional con 2048 elementos. Cada elemento de ese vector contendrá el valor numérico que representa una característica particular extraída de la imagen.

En resumen, las características extraídas se almacenan como valores numéricos en un vector unidimensional de 2048 elementos en la lista features_list. Estos valores representan las características de alto nivel de las imágenes, y su significado específico depende de cómo se entrenó el modelo (en este caso, ResNet50) y las características que ha aprendido a identificar.

Ahora en el codigo estaremos agregando las caracteristicas extraidas a un dataframe de Pandas, para poder luego separar bien nuestros datos que seran entrenados, y tambien nos encargaremos de tener bien etiquetados los datos:

```
[ ]  features_df = pd.DataFrame(features_list)

[ ]  features_df["labels"] = data["labels"]

[ ]  x = features_df.drop(['labels'], axis = 1)
     y = features_df.loc[:,"labels"].values

[ ]  x

[ ]  print(f"Number of features before feature selection: {x.shape[1]}")

[ ]  y
```

features_df = pd.DataFrame(features_list): Se crea un DataFrame de Pandas llamado features_df a partir de la lista features_list. El DataFrame contendrá las características extraídas de las imágenes.

features_df["labels"] = data["labels"]: Se agrega una columna adicional al DataFrame features_df llamada "labels" que contiene las etiquetas de clase de las imágenes. Estas etiquetas se obtienen del DataFrame data que contiene información sobre si una imagen representa una célula sana (etiqueta 0) o una célula con leucemia (etiqueta 1).

x = features_df.drop(['labels'], axis=1): Se crea un nuevo DataFrame x que contiene todas las columnas de features_df excepto la columna "labels". En otras palabras, x almacena solo las características extraídas de las imágenes.

```
x
```

	0	1	2	3	4	5	6	7
0	3.888962	0.003357	0.000000	0.080566	0.002496	0.003216	0.178870	0.05421
1	6.105570	0.000000	0.008155	0.063793	0.005606	0.047762	0.079004	0.00000
2	4.191668	0.000000	0.000000	0.073728	0.007410	0.040819	0.126977	0.02611
3	4.969845	0.190751	0.000000	0.060903	0.012210	0.034154	0.038725	0.03580
4	7.409761	0.000000	0.000000	0.223266	0.000000	0.013826	0.060580	0.00000
...	...	...	...	...	...	...	...	...
10656	4.650620	0.000000	0.000000	0.003782	0.000000	0.000000	0.079633	0.03596
10657	5.075252	0.000000	0.000000	0.073403	0.000000	0.050068	0.109488	0.00000
10658	5.491915	0.000000	0.000000	0.062658	0.087210	0.227873	0.310514	0.03740
10659	4.076107	0.017551	0.000000	0.118688	0.000000	0.000000	0.148364	0.00000
10660	8.373825	0.000000	0.038963	0.043950	0.000000	0.000000	0.222565	0.00000

10661 rows × 2048 columns

Este seria un ejemplo de lo que ocurre cuando se imprime la variable x, la cual es un dataframe construido a base del DF features_df pero sin los labels, que vienen a ser las etiquetas 1 y 0.
Es decir, solo contiene las 2.046 características de las 10.661 imágenes.

y = features_df.loc[:,"labels"].values: Se crea un arreglo NumPy llamado y que almacena las etiquetas de clase (0 o 1) en forma de valores numéricos. Estas etiquetas corresponden a la presencia de leucemia (1) o la ausencia de leucemia (0) en las imágenes.

x: Esta línea no realiza una acción específica, pero muestra el contenido de x, que es el DataFrame que contiene las características extraídas de las imágenes.

print(f"Number of features before feature selection: {x.shape[1]}"): Esta línea imprime en la consola el número de características antes de realizar alguna selección de características. Utiliza el atributo .shape de x para obtener el número de columnas, que representa el número de características.

y: Esta línea muestra en la consola el contenido del arreglo y, que contiene las etiquetas de clase en forma de valores numéricos (0 o 1). Este arreglo se utiliza

posteriormente como las etiquetas objetivo en un modelo de aprendizaje automático. Este es un ejemplo de lo que ocurre cuando imprimimos y:

```
In [20]:
y

Out[20]:
array([1, 1, 1, ..., 0, 0, 0])
```

Con lo que añadimos ahora tenemos que el código agrega etiquetas de clase a las características extraídas y organiza los datos en un DataFrame de Pandas (features_df). Luego, crea dos conjuntos de datos, x que contiene las características y y que contiene las etiquetas de clase en forma de valores numéricos. x se utiliza para entrenar un modelo de aprendizaje automático, mientras que "y" representa las etiquetas objetivo del modelo, recordemos que las etiquetas son 1 para las celulas con ALL y 0 para las HEM (Células sin leucemia).

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(x)
x_ = scaler.transform(x)

x_ = pd.DataFrame(x_)

from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif

def anova_fs():

    selector = SelectKBest(f_classif, k=500) # k is number of features
    selector.fit(x_, y)

    cols = selector.get_support(indices=True)
    anova_x = x_[cols]
    return anova_x
```

from sklearn.preprocessing import MinMaxScaler: Se importa la clase MinMaxScaler del módulo sklearn.preprocessing. Esta clase se utiliza para escalar las características de manera que estén en un rango específico, en este caso, el rango [0, 1].

scaler = MinMaxScaler(): Se crea una instancia del objeto MinMaxScaler llamada scaler. Esta instancia se utilizará para ajustar y transformar las características.

scaler.fit(x): Se ajusta el escalador scaler a los datos de características x. Esto calcula los valores mínimos y máximos de cada característica en el conjunto de datos x y los utiliza para escalar las características.

x_ = scaler.transform(x): Se utiliza el escalador scaler para transformar las características en x. Las características transformadas se almacenan en un nuevo DataFrame llamado x_.

x_ = pd.DataFrame(x_): El resultado de la transformación se convierte en un DataFrame de Pandas llamado x_. Esto se hace para asegurarse de que los datos estén en un formato de DataFrame.

from sklearn.feature_selection import SelectKBest y from sklearn.feature_selection import f_classif: Se importan las clases SelectKBest y f_classif del módulo sklearn.feature_selection. Estas clases se utilizan para realizar selección de características mediante análisis de varianza (ANOVA).

def anova_fs(): Se define una función llamada anova_fs que realizará la selección de características utilizando ANOVA.

selector = SelectKBest(f_classif, k=500): Se crea una instancia del objeto SelectKBest con la función de prueba ANOVA f_classif y se establece k en 500, lo que significa que se seleccionarán las mejores 500 características según el análisis de varianza.

selector.fit(x_, y): El selector se ajusta a los datos de características x_ y las etiquetas de clase y. Esto realiza el cálculo de ANOVA para evaluar la importancia de cada característica en relación con la variable objetivo.

cols = selector.get_support(indices=True): Se obtienen los índices de las características seleccionadas por ANOVA. cols contiene los índices de las características más importantes.

anova_x = x_[cols]: Se crea un nuevo DataFrame llamado anova_x que contiene solo las características seleccionadas por ANOVA. Estas características se almacenan en anova_x y se utilizan para tareas posteriores.

return anova_x: La función devuelve el DataFrame anova_x que contiene las características seleccionadas por el análisis de varianza.

En resumen, este código realiza un preprocesamiento de características, escalándolas con MinMaxScaler, y luego realiza la selección de características utilizando el análisis de

varianza (ANOVA) a través de SelectKBest. La función anova_fs devuelve las características seleccionadas por ANOVA.

Originalmente, tenaimos 2048 características en el DataFrame x. Luego, después de realizar la selección de características mediante el análisis de varianza (ANOVA) con SelectKBest, se estableció k en 500. Esto significa que ahora deberiamos tener 500 características seleccionadas en el DataFrame anova_x. Por lo tanto, el número de características se ha reducido de 2048 a 500 después de aplicar la selección de características.

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier

def RFE_fs():
    rfe_selector = RFE(estimator=RandomForestClassifier())
    rfe_selector.fit(x_, y)

    rfe_support = rfe_selector.get_support()
    rfe_feature = x_.loc[:,rfe_support].columns.tolist()

    rfe_x = x_[rfe_feature]
    return rfe_x
```

from sklearn.feature_selection import RFE y from sklearn.ensemble import RandomForestClassifier: Se importa la clase RFE del módulo sklearn.feature_selection y la clase RandomForestClassifier del módulo sklearn.ensemble. RFE se utiliza para realizar la eliminación recursiva de características, mientras que RandomForestClassifier se utiliza como el estimador de modelos de bosques aleatorios.

def RFE_fs(): Se define una función llamada RFE_fs que realizará la selección de características utilizando Recursive Feature Elimination (RFE) con Random Forest.

rfe_selector = RFE(estimator=RandomForestClassifier()): Se crea una instancia del objeto RFE con RandomForestClassifier como estimador. Esto significa que se utilizará un clasificador de Bosques Aleatorios para evaluar la importancia de cada característica.

rfe_selector.fit(x_, y): El selector RFE se ajusta a los datos de características x_ y las etiquetas de clase y. Realiza un proceso iterativo en el que elimina recursivamente las

características menos importantes y mantiene las más importantes según la evaluación del clasificador de Bosques Aleatorios.

rfe_support = rfe_selector.get_support(): Se obtiene una máscara de soporte que indica qué características han sido seleccionadas como importantes por RFE. rfe_support es una lista de booleanos donde True indica que la característica correspondiente se ha seleccionado, y False indica que no se ha seleccionado.

rfe_feature = x_.loc[:, rfe_support].columns.tolist(): Se obtienen las columnas del DataFrame x_ que tienen True en la máscara de soporte. Estas son las características seleccionadas por RFE. rfe_feature es una lista de nombres de columnas que representan las características seleccionadas.

rfe_x = x_[rfe_feature]: Se crea un nuevo DataFrame llamado rfe_x que contiene solo las características seleccionadas por RFE. Estas características se almacenan en rfe_x y se utilizan para tareas posteriores.

return rfe_x: La función devuelve el DataFrame rfe_x, que contiene las características seleccionadas por Recursive Feature Elimination.

En resumen, este código utiliza Recursive Feature Elimination (RFE) con un clasificador de Bosques Aleatorios para seleccionar un subconjunto de características más importantes a partir del conjunto original de características. El resultado es un DataFrame llamado rfe_x que contiene solo las características seleccionadas por RFE.

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier

def rf_fs():
    embeded_rf_selector = SelectFromModel(RandomForestClassifier(n_estimators=200, random_state=5), threshold='1.25*med
    embeded_rf_selector.fit(x, y)

    embeded_rf_support = embeded_rf_selector.get_support()
    embeded_rf_feature = x.loc[:,embeded_rf_support].columns.tolist()

    rf_x = x[embeded_rf_feature]
    return rf_x
```

Este fragmento de código introduce una estrategia de selección de características llamada "Select From Model" (Seleccionar Desde el Modelo) que utiliza un clasificador de Bosques Aleatorios. Aquí está una descripción línea por línea:

from sklearn.feature_selection import SelectFromModel y from sklearn.ensemble import RandomForestClassifier: Se importa la clase SelectFromModel del módulo sklearn.feature_selection y la clase RandomForestClassifier del módulo sklearn.ensemble. SelectFromModel se utilizará para realizar la selección de características basada en un modelo, y RandomForestClassifier se utilizará como el modelo.

def rf_fs(): Se define una función llamada rf_fs que realizará la selección de características utilizando "Select From Model" con un clasificador de Bosques Aleatorios.

embeded_rf_selector = SelectFromModel(RandomForestClassifier(n_estimators=200, random_state=5), threshold='1.25*median'): Se crea una instancia del objeto SelectFromModel con un clasificador de Bosques Aleatorios configurado con 200 estimadores (n_estimators=200) y una semilla aleatoria (random_state=5). La threshold se establece en '1.25*median', lo que significa que las características cuya importancia es mayor que 1.25 veces la mediana de importancias se seleccionarán.

embeded_rf_selector.fit(x, y): El selector de características basado en Bosques Aleatorios se ajusta a los datos de características x y las etiquetas de clase y. Evalúa la importancia de cada característica utilizando el modelo de Bosques Aleatorios y selecciona las características que cumplen con el criterio de umbral especificado.

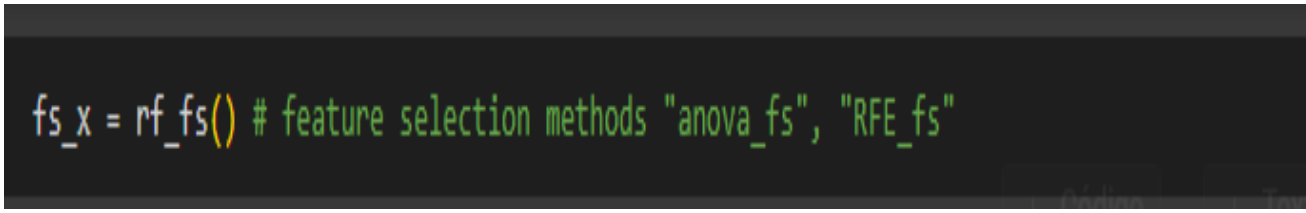
embeded_rf_support = embeded_rf_selector.get_support(): Se obtiene una máscara de soporte que indica qué características han sido seleccionadas como importantes por el método "Select From Model". embeded_rf_support es una lista de booleanos donde True indica que la característica correspondiente se ha seleccionado, y False indica que no se ha seleccionado.

embeded_rf_feature = x.loc[:, embeded_rf_support].columns.tolist(): Se obtienen las columnas del DataFrame x que tienen True en la máscara de soporte. Estas son las características seleccionadas por el método "Select From Model". embeded_rf_feature es una lista de nombres de columnas que representan las características seleccionadas.

rf_x = x[embeded_rf_feature]: Se crea un nuevo DataFrame llamado rf_x que contiene solo las características seleccionadas por el método "Select From Model". Estas características se almacenan en rf_x y se utilizan para tareas posteriores.

return rf_x: La función devuelve el DataFrame rf_x, que contiene las características seleccionadas por el método "Select From Model".

En resumen, este código utiliza un clasificador de Bosques Aleatorios para realizar la selección de características basada en la importancia de características y el umbral de mediana especificado. Las características seleccionadas se almacenan en un DataFrame llamado rf_x.



La línea de código fs_x = rf_fs() está asignando el resultado de la función rf_fs() a la variable fs_x

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(fs_x, y, test_size = 0.2, random_state = 42)
```

la línea completa imprimirá un mensaje similar a "Número de características después de la selección de características: N", donde "N" es el número real de características en el DataFrame fs_x después de la selección de características. Esto proporciona información sobre cuántas características se han mantenido después del proceso de selección.

```
print(f"Number of features after feature selection: {fs_x.shape[1]}")
```

```
Number of features after feature selection: 584
```

Este era un ejemplo de que sucederia si es que ejecutamos esa linea despues de haber pasado por el proceso de selección de caracteristicas.

Ahora pasamos con el Train-test Split:

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(fs_x, y, test_size = 0.2, random_state = 42)

from sklearn.model_selection import cross_val_score,cross_val_predict
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn import svm
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score,precision_score,recall_score,accuracy_score
from sklearn.model_selection import GridSearchCV
```

A continuación, explicaré cada una de las importaciones y su utilidad en la clasificación de imágenes:

from sklearn.model_selection import train_test_split: Esta importación proviene del módulo sklearn.model_selection **y se utiliza para dividir el conjunto de datos en conjuntos de entrenamiento y prueba**. La función **train_test_split** permite crear automáticamente subconjuntos aleatorios para entrenar y evaluar modelos de aprendizaje automático.

x_train, x_test, y_train, y_test = train_test_split(fs_x, y, test_size=0.2, random_state=42): Aquí, se utiliza train_test_split para dividir las características (fs_x) y las etiquetas (y) en conjuntos de entrenamiento y prueba. test_size establece la proporción de datos de prueba, y random_state garantiza la reproducibilidad de la división.

from sklearn.model_selection import cross_val_score, cross_val_predict: Estas importaciones se utilizan para realizar validación cruzada. cross_val_score calcula las puntuaciones de validación cruzada para un modelo, y cross_val_predict genera predicciones en validación cruzada.

from sklearn.ensemble import RandomForestClassifier: El RandomForestClassifier es un clasificador basado en Bosques Aleatorios, que es un conjunto de árboles de decisión. Se utiliza para clasificación y es especialmente eficaz para problemas de clasificación en imágenes.

from sklearn.naive_bayes import GaussianNB: El GaussianNB es un clasificador Bayesiano ingenuo que asume que las características son independientes y siguen una distribución normal. Aunque no es comúnmente utilizado en la clasificación de imágenes, es eficaz en algunas situaciones.

from sklearn.neighbors import KNeighborsClassifier: El KNeighborsClassifier es un clasificador basado en vecinos más cercanos. Se utiliza para clasificar datos basados en la similitud con sus vecinos más cercanos y es adecuado para problemas de clasificación en imágenes.

from sklearn import svm: La importación svm se refiere a las máquinas de soporte vectorial (SVM). Las SVM son algoritmos de clasificación poderosos y ampliamente utilizados en la clasificación de imágenes. Pueden manejar problemas de clasificación binaria y multiclase.

from sklearn.metrics import confusion_matrix, f1_score, precision_score, recall_score, accuracy_score: Estas importaciones son para métricas de evaluación de modelos. confusion_matrix calcula la matriz de confusión que muestra la calidad de las predicciones. f1_score, precision_score, recall_score y accuracy_score son métricas comunes para medir el rendimiento del modelo en términos de precisión, recuperación, puntaje F1 y precisión.

from sklearn.model_selection import GridSearchCV: Esta importación es para la búsqueda de hiperparámetros a través de validación cruzada. GridSearchCV permite explorar diferentes combinaciones de hiperparámetros para encontrar los mejores valores para un modelo.

Estas importaciones proporcionan herramientas y modelos para la clasificación de imágenes, evaluación del rendimiento del modelo y optimización de hiperparámetros. Permiten realizar divisiones de datos, validación cruzada, entrenar varios clasificadores y evaluar su rendimiento en términos de métricas de calidad del modelo. Cada una de estas herramientas y modelos tiene un papel específico en la construcción y evaluación de modelos de clasificación de imágenes.

Hasta aquí llega la documentación del procesamiento de los datos, la etiquetación y selección de características, de ahora en adelante vamos a estar documentando todos los modelos de clasificación que vamos a estar utilizando en este proyecto:

kNN:

```
neig = np.arange(1, 25)
train_accuracy = []
test_accuracy = []

for i, k in enumerate(neig):

    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(x_train,y_train)
    prediction_ = knn.predict(x_test)
    train_accuracy.append(knn.score(x_train, y_train))
    test_accuracy.append(knn.score(x_test, y_test))

print("Best accuracy is {} with K = {}".format(np.max(test_accuracy),1+test_accuracy.index(np.max(test_accuracy))))
```

```
knn = KNeighborsClassifier(n_neighbors=17)
knn.fit(x_train,y_train)
predicted = knn.predict(x_test)
score = knn.score(x_test, y_test)
knn_score_ = np.mean(score)

print('Accuracy : %.3f' % (knn_score_))
```

Este codigo se utiliza para entrenar un modelo de clasificación llamado k-Nearest Neighbors (kNN) y evaluar su rendimiento. Aquí está una descripción línea por línea:

neig = np.arange(1, 25): Se crea un arreglo NumPy llamado neig que contiene valores enteros desde 1 hasta 24. Estos valores representan el número de vecinos más cercanos (k) que se utilizarán en el clasificador kNN.

train_accuracy = [] y test_accuracy = []: Se inicializan dos listas vacías, train_accuracy y test_accuracy, que se utilizarán para almacenar las puntuaciones de precisión del modelo en los conjuntos de entrenamiento y prueba.

El bucle for itera a través de los valores de neig:

for i, k in enumerate(neig): i es el índice del valor actual de k, y k es el número de vecinos más cercanos que se está probando en esta iteración.

knn = KNeighborsClassifier(n_neighbors=k): Se crea un clasificador kNN con el número de vecinos especificado por k.

knn.fit(x_train, y_train): Se entrena el clasificador kNN en el conjunto de entrenamiento (x_train, y_train).

prediction_ = knn.predict(x_test): Se realizan predicciones en el conjunto de prueba (x_test) utilizando el clasificador kNN entrenado.

train_accuracy.append(knn.score(x_train, y_train)) y test_accuracy.append(knn.score(x_test, y_test)): Se calculan y almacenan las puntuaciones de precisión del clasificador kNN en los conjuntos de entrenamiento y prueba, respectivamente.

Después del bucle for, se imprime el valor de K (número de vecinos) que produce la mejor precisión en el conjunto de prueba: "Best accuracy is {} with K = {}".format(np.max(test_accuracy), 1 + test_accuracy.index(np.max(test_accuracy))).

Se crea un nuevo clasificador kNN con el número de vecinos óptimo, que en este caso es 17, y se entrena en el conjunto de entrenamiento.

Se realizan predicciones en el conjunto de prueba y se calcula la precisión del modelo: knn.score(x_test, y_test).

La precisión del modelo se almacena en knn_score_.

Finalmente, se imprime la precisión del modelo en el conjunto de prueba: 'Accuracy : %.3f' % (knn_score_).

Lo que hace este código es que entrena un clasificador kNN con diferentes valores de K (número de vecinos) y determina cuál produce la mejor precisión en el conjunto de prueba. Luego, se informa la precisión del modelo con el valor óptimo de K.

```
Accuracy : 0.834
```

Luego tenemos las siguientes lineas de aquí:

```
p=precision_score(y_test, predicted)
print('Precision : %.3f' % (p))

r=recall_score(y_test, predicted)
print('Recall : %.3f' % (r))

f1=f1_score(y_test, predicted)
print('F1-score: %.3f' % (f1))

f1_w=f1_score(y_test, predicted, average='weighted')
print('Weighted f1-score: %.3f' % (f1_w))

cf_matrix = confusion_matrix(y_test, predicted)
sns.heatmap(cf_matrix, cmap="PuBu", annot=True, fmt='.0f')
plt.show()
```

Estas líneas de código se utilizan para calcular y mostrar métricas de evaluación del modelo, así como para visualizar la matriz de confusión. Aquí está una descripción de cada línea:

p = precision_score(y_test, predicted): Calcula la puntuación de precisión del modelo. La precisión mide la proporción de predicciones positivas que son correctas. El resultado se almacena en la variable p.

print('Precision : %.3f' % (p)): Imprime la puntuación de precisión con un formato específico en la consola.

r = recall_score(y_test, predicted): Calcula la puntuación de recuperación (recall) del modelo. La recuperación mide la proporción de ejemplos positivos reales que se han clasificado correctamente como positivos. El resultado se almacena en la variable r.

print('Recall : %.3f' % (r)): Imprime la puntuación de recuperación en la consola.

f1 = f1_score(y_test, predicted): Calcula el puntaje F1 del modelo. El puntaje F1 es una medida que combina precisión y recuperación en una única métrica. El resultado se almacena en la variable f1.

print('F1-score: %.3f' % (f1)): Imprime el puntaje F1 en la consola.

f1_w = f1_score(y_test, predicted, average='weighted'): Calcula el puntaje F1 ponderado. El puntaje F1 ponderado tiene en cuenta el desequilibrio de clases en los datos. El resultado se almacena en la variable f1_w.

print('Weighted f1-score: %.3f' % (f1_w)): Imprime el puntaje F1 ponderado en la consola.

cf_matrix = confusion_matrix(y_test, predicted): Calcula la matriz de confusión que muestra las predicciones del modelo en comparación con las etiquetas reales. El resultado se almacena en la variable cf_matrix.

sns.heatmap(cf_matrix, cmap="PuBu", annot=True, fmt='.0f'): Visualiza la matriz de confusión como un mapa de calor (heatmap) utilizando la biblioteca Seaborn (sns). Esto proporciona una representación gráfica de las predicciones del modelo.

plt.show(): Muestra el mapa de calor en una ventana gráfica.

En resumen, estas líneas de código calculan y muestran varias métricas de evaluación del modelo, como precisión, recuperación y puntaje F1, además de visualizar la matriz de confusión para comprender el rendimiento del modelo en la clasificación de imágenes.

```
Precision : 0.841
Recall : 0.933
F1-score: 0.885
Weighted f1-score: 0.827
```

Ahora hacemos lo mismo, pero con el modelo: SVM

```
param_grid_svm = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['rbf', 'poly']}

SVM_grid = GridSearchCV(svm.SVC(), param_grid_svm, cv=5)
SVM_grid.fit(x_train, y_train)

print(SVM_grid.best_params_)

print(SVM_grid.best_estimator_)

svm_clf = svm.SVC(C=100, gamma=0.01, kernel='rbf')
svm_clf.fit(x_train,y_train)
predicted = svm_clf.predict(x_test)
score = svm_clf.score(x_test, y_test)
svm_score_ = np.mean(score)

print('Accuracy : %.3f' % (svm_score_))
```

param_grid_svm: Define un diccionario llamado param_grid_svm que contiene los hiperparámetros que se van a explorar en la búsqueda de cuadrícula. En este caso, se están buscando diferentes valores para el parámetro C, gamma, y kernel. Se considerarán varias combinaciones de estos valores.

SVM_grid = GridSearchCV(svm.SVC(), param_grid_svm, cv=5): Crea un objeto GridSearchCV que se utilizará para buscar los mejores hiperparámetros para un modelo SVM. Se utiliza svm.SVC() como estimador base y se proporciona el diccionario param_grid_svm como conjunto de hiperparámetros para explorar. cv=5 indica que se utilizará una validación cruzada de 5 pliegues.

SVM_grid.fit(x_train, y_train): Realiza la búsqueda de cuadrícula para encontrar los mejores hiperparámetros. Entrena y evalúa el modelo SVM en el conjunto de entrenamiento (x_train, y_train) para cada combinación de hiperparámetros.

print(SVM_grid.best_params_): Imprime los mejores hiperparámetros encontrados por la búsqueda de cuadrícula. Estos son los valores óptimos de C, gamma y kernel.

print(SVM_grid.best_estimator_): Imprime el estimador SVM completo con los mejores hiperparámetros encontrados. Esto proporciona un modelo SVM óptimo.

svm_clf = svm.SVC(C=100, gamma=0.01, kernel='rbf'): Crea un nuevo modelo SVM con los hiperparámetros óptimos encontrados por la búsqueda de cuadrícula.

svm_clf.fit(x_train, y_train): Entrena el modelo SVM con los hiperparámetros óptimos en el conjunto de entrenamiento.

predicted = svm_clf.predict(x_test): Realiza predicciones en el conjunto de prueba utilizando el modelo SVM entrenado.

score = svm_clf.score(x_test, y_test): Calcula la precisión del modelo SVM en el conjunto de prueba.

svm_score_ = np.mean(score): Calcula la precisión media del modelo SVM en el conjunto de prueba.

print('Accuracy : %.3f' % (svm_score_)): Imprime la precisión media del modelo SVM en la consola.

En resumen, este código realiza una búsqueda de hiperparámetros para encontrar los valores óptimos de C, gamma y kernel para un modelo SVM. Luego, entrena un modelo SVM con los hiperparámetros óptimos y evalúa su precisión en el conjunto de prueba. Las métricas de rendimiento se imprimen en la consola.

```
Accuracy : 0.900
```

Luego de eso se realiza lo mismo con la matriz de confusión y estos son los demas datos optenidos por el modelo, lo que nos interesa aquí es ver que modelo tiene mayor accuracy para poder usarlo en nuestra aplicación:

```
precision : 0.902
recall : 0.957
f1-score: 0.929
weighted f1-score: 0.898
```

Ahora tenemos el Random Forest:

```
param_grid_rf = {
    'n_estimators': [200, 500],
    'max_depth' : [4,5,6,7,8]}

RF_grid = GridSearchCV(estimator=RandomForestClassifier(), param_grid=param_grid_rf, cv= 5)
RF_grid.fit(x_train, y_train)

print(RF_grid.best_params_)

r_forest = RandomForestClassifier(500,max_depth=8, random_state=5)
r_forest.fit(x_train,y_train)
predicted = r_forest.predict(x_test)
score = r_forest.score(x_test, y_test)
rf_score_ = np.mean(score)

print('Accuracy : %.3f' % (rf_score_))
```

Estas líneas de código se utilizan para realizar una búsqueda de hiperparámetros y entrenar un modelo de Clasificación con Bosques Aleatorios (Random Forest) en un problema de clasificación. Aquí está una descripción de cada línea:

param_grid_rf: Define un diccionario llamado param_grid_rf que contiene los hiperparámetros que se van a explorar en la búsqueda de cuadrícula. En este caso, se están buscando diferentes valores para el número de estimadores (n_estimators) y la profundidad máxima del árbol (max_depth). Se considerarán varias combinaciones de estos valores.

RF_grid = GridSearchCV(estimator=RandomForestClassifier(), param_grid=param_grid_rf, cv=5): Crea un objeto GridSearchCV que se utilizará para buscar los mejores hiperparámetros para un modelo de Bosques Aleatorios. Se utiliza RandomForestClassifier() como estimador base y se proporciona el diccionario param_grid_rf como conjunto de hiperparámetros para explorar. cv=5 indica que se utilizará una validación cruzada de 5 pliegues.

RF_grid.fit(x_train, y_train): Realiza la búsqueda de cuadrícula para encontrar los mejores hiperparámetros. Entrena y evalúa el modelo de Bosques Aleatorios en el conjunto de entrenamiento (x_train, y_train) para cada combinación de hiperparámetros.

print(RF_grid.best_params_): Imprime los mejores hiperparámetros encontrados por la búsqueda de cuadrícula. Estos son los valores óptimos de n_estimators y max_depth.

r_forest = RandomForestClassifier(500, max_depth=8, random_state=5): Crea un nuevo modelo de Bosques Aleatorios con los hiperparámetros óptimos encontrados por la búsqueda de cuadrícula.

r_forest.fit(x_train, y_train): Entrena el modelo de Bosques Aleatorios con los hiperparámetros óptimos en el conjunto de entrenamiento.

predicted = r_forest.predict(x_test): Realiza predicciones en el conjunto de prueba utilizando el modelo de Bosques Aleatorios entrenado.

score = r_forest.score(x_test, y_test): Calcula la precisión del modelo de Bosques Aleatorios en el conjunto de prueba.

rf_score_ = np.mean(score): Calcula la precisión media del modelo de Bosques Aleatorios en el conjunto de prueba.

print('Accuracy : %.3f' % (rf_score_)): Imprime la precisión media del modelo de Bosques Aleatorios en la consola.

```
Accuracy : 0.826
```

```
precision : 0.821
recall : 0.953
f1-score: 0.882
weighted f1-score: 0.814
```

Ahora tenemos el Naive Bayes:

```
nb_model = GaussianNB()
nb_model.fit(x_train,y_train)
predicted = nb_model.predict(x_test)
score = nb_model.score(x_test, y_test)
nb_score_ = np.mean(score)

print('Accuracy : %.3f' % (nb_score_))
```

Este fragmento de código se utiliza para entrenar y evaluar un modelo de Clasificador Naive Bayes (Gaussian Naive Bayes). Aquí tienes una descripción de cada línea:

nb_model = GaussianNB(): Se crea un objeto de modelo de Clasificador Naive Bayes utilizando la clase GaussianNB. En este caso, se está utilizando el clasificador Naive Bayes Gaussiano, que es adecuado para datos continuos donde se asume que las características siguen una distribución normal.

nb_model.fit(x_train, y_train): Se entrena el modelo Naive Bayes con los datos de entrenamiento (x_train, y_train). El modelo aprenderá a hacer predicciones basadas en las características de entrada y las etiquetas de entrenamiento.

predicted = nb_model.predict(x_test): Se realizan predicciones en el conjunto de prueba (x_test) utilizando el modelo Naive Bayes entrenado.

score = nb_model.score(x_test, y_test): Se calcula la precisión del modelo Naive Bayes en el conjunto de prueba. La precisión mide la proporción de predicciones correctas.

nb_score_ = np.mean(score): Calcula la precisión media del modelo Naive Bayes en el conjunto de prueba. En este caso, la precisión media es el mismo valor que score debido a que el modelo Naive Bayes no produce múltiples resultados en una evaluación.

print('Accuracy : %.3f' % (nb_score_)): Imprime la precisión media del modelo Naive Bayes en la consola.

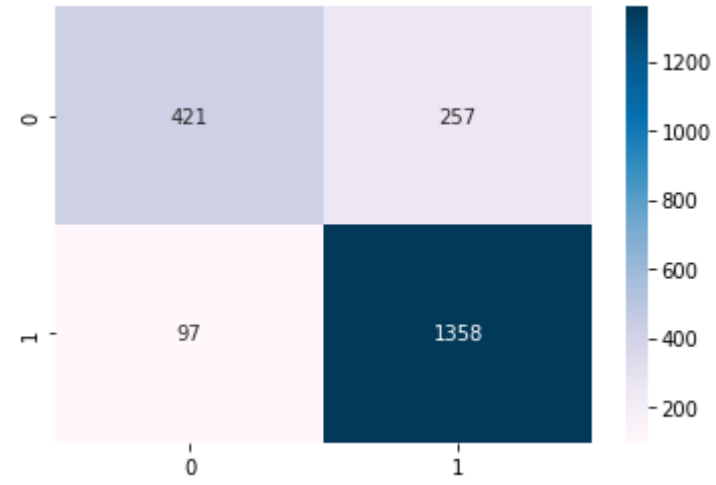
En resumen, este código entrena un modelo de Clasificador Naive Bayes Gaussiano en el conjunto de entrenamiento y calcula su precisión en el conjunto de prueba. Luego, se imprime la precisión del modelo en la consola.

La cual fue de:

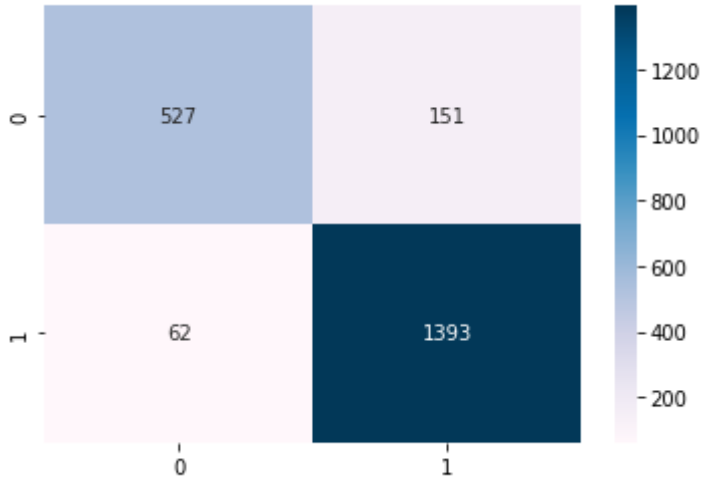
```
Accuracy : 0.767


precision : 0.852
recall : 0.798
f1-score: 0.824
weighted f1-score: 0.771
```

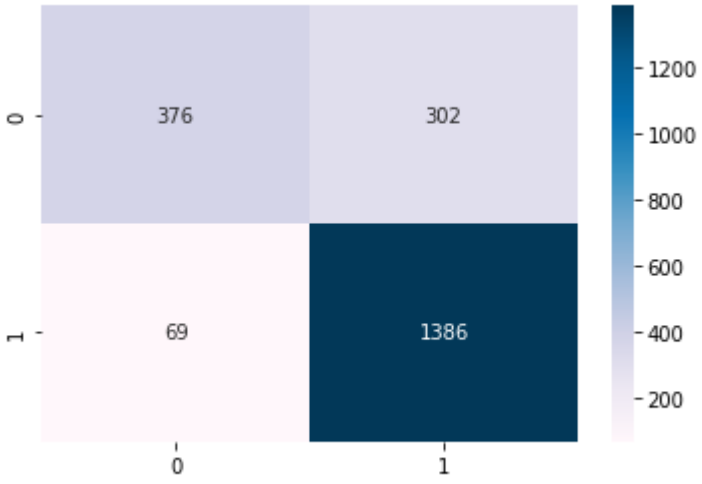
Ahora estaremos viendo la Matriz de confusion de cada uno de los 4 modelos de clasificacion kNN:



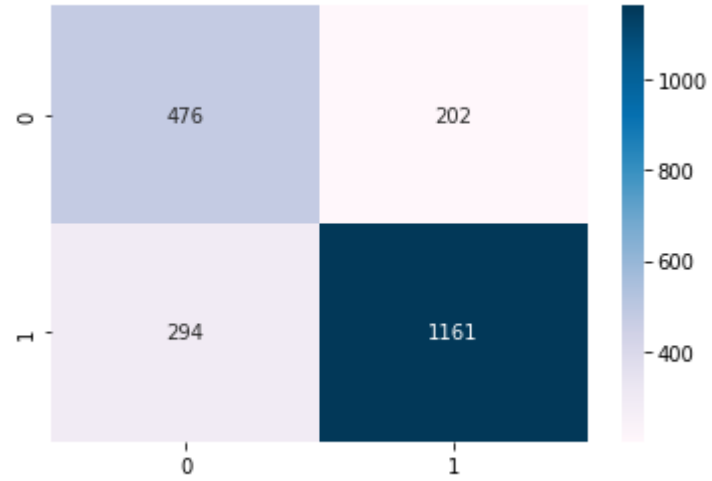
SVM:



Random Forest:



Naive bayes:



Como última instancia nos fijamos en que efectivamente el modelo de clasificación SVM, que fue el que mayor accuracy tuvo se guarda correctamente en un archivo .H5 que lo que hace es guardar el modelo que posteriormente lo usaremos haciendo uso de la herramienta de import model de sklearn.