Desafíos de Programación

Autor: Antonio Martínez Santa
Materia: Desafíos de Programación
DNI: 77599131Y
Centro: Universidad de Alicante
Fecha: 06/12/2024

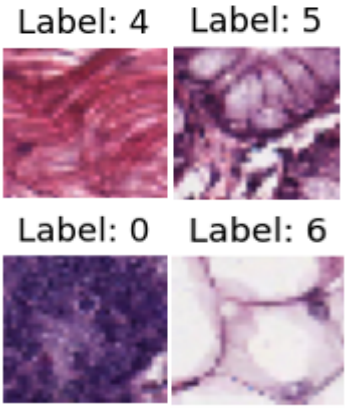
Índice

Capítulo I : Clasificación de imágenes de histologías colorrectales

Objetivo

El principal propósito de esta práctica es desarrollar una aplicación basada en redes neuronales de Deep Learning para lograr así determinar, entre un total de 5000 imágenes de histologías colorrectales, la categoría a la que pertenece. En nuestro caso tenemos un total de 8 categorías distribuidas entre las siguientes:

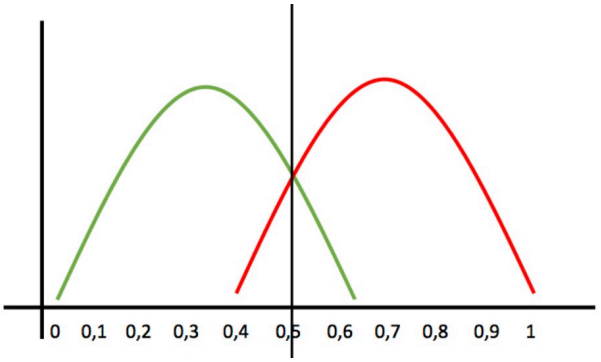
- 0: TUMOR
- 1: STROMA
- 2: COMPLEX
- 3: LYMPHO
- 4: DEBRIS
- 5: MUCOSA
- 6: ADIPOSE
- 7: EMPTY



El objetivo es mejorar la propia red convolucional que hay propuesta para así lograr incrementar su valor de AUC en igualdad de condiciones, ya que a mayor AUC indica que nuestro clasificador sabe diferenciar más los conjuntos de datos que maneja, pero…

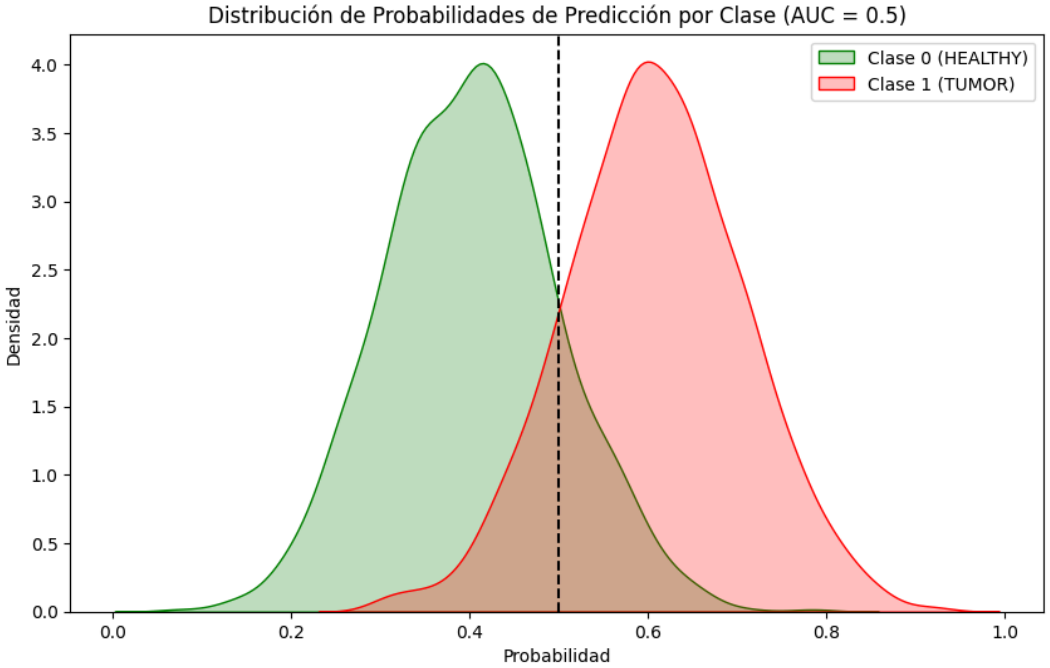
¿Qué es AUC?

El AUC (Area Under Curve) representa el área bajo la curva ROC mide el desempeño que tiene el modelo, en este caso específico nos representará qué tan bien nuestro modelo es capaz de distinguir entre una imagen que sí es TUMOR con una imagen que no lo es.



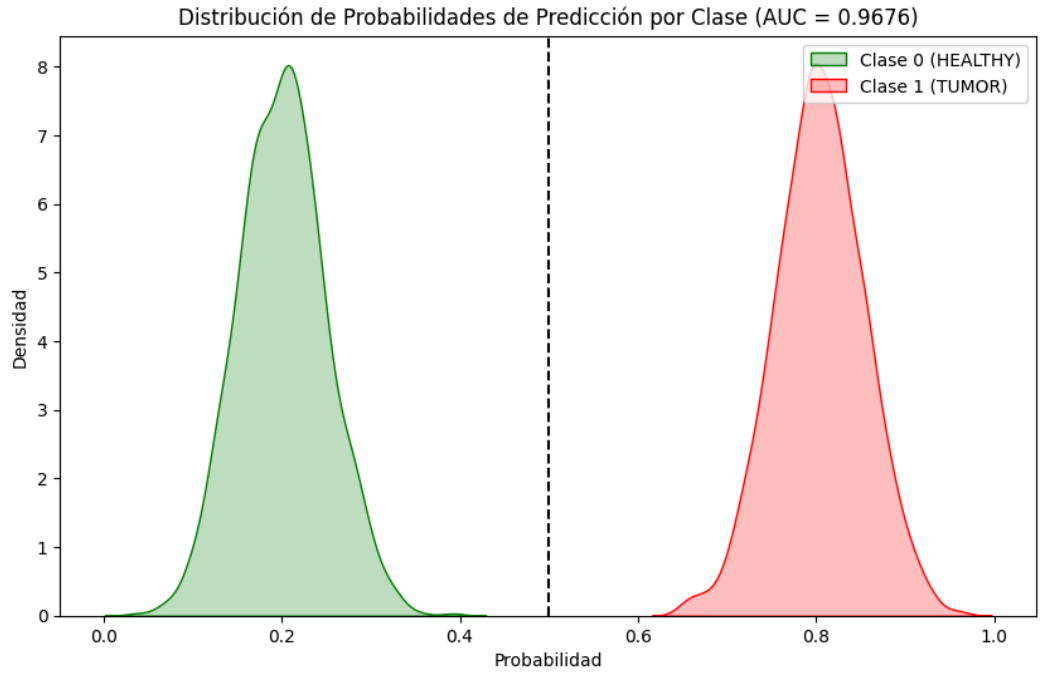
Si estos dos conjuntos son muy cercanos entre sí indica que las predicciones no son lo suficientemente correctas ya que está mezclando las predicciones de cada clase, estos valores de AUC rondan los valores cercanos a 0.5.

La gráfica asociada a este valor de AUC sería algo por este estilo:



Como podemos observar muchas predicciones se juntan en el punto central de ambas curvas, esto lleva a una muy mala discriminicación de los datos en cuanto a las 2 clases que tenemos.

En caso de que estén muy alejadas indica que claramente está discriminando correctamente ambos conjuntos de predicciones ya que hay una clara diferencia entre cada clase, estos valores rondan los valores cercanos a 1, esto es señal de un buen clasificador. La gráfica asociada a un caso particular de un valor cercano a 1 sería la siguiente (destacar que este valor de AUC fue obtenido al aplicar 1 vez 10-CV, no es el mejor obtenido):



Aquí podemos apreciar una total heterogeneidad de las predicciones, es decir, nuestro clasificador está diferenciando bien ambas clases.

Problema base

Como punto de partida se nos ofrece un código el cuál daba como resultado un valor de AUC de 0.5, esto muestra la incapacidad del modelo para clasificar adecuadamente cada una de las imágenes.

```
AUC 0.5000
```

Como mencioné anteriormente debemos crear (o adaptar) este modelo a uno que sí sea capaz de clasificar cada una de estas imágenes con su correspondiente.
Además se nos proporciona más información con respecto al modelo inicial que son la matriz de confusión y las métricas Precision, Recall y F1 de TUMOR y de HEALTHY.

Matriz de confusión

En esta matriz de confusión las filas representan las etiquetas verdaderas, es decir, lo que realmente es cada una de las muestras, mientras que las columnas representan las predicciones del modelo.

```
Confusion matrix
[[   0  155]
 [   0 1095]]
```

Aplicado a lo proporcionado por el modelo obtenemos una tabla de esta forma:

	Predicho TUMOR (Clase 0)	Predicho HEALTHY (Clase 1)	Descripción
TUMOR	0	155	**Falsos Negativos:** Muestras que predijo que eran HEALTHY cuando realmente eran TUMOR
HEALTHY	0	1095	**Verdaderos Positivos:** Muestras que son realmente HEALTHY y el modelo predijo HEALTHY

Analizando estos resultados podemos obtener las siguientes conclusiones:

- Primera fila:
  - Predicho TUMOR (Clase 0): El modelo ha predicho que todas las imágenes son HEALTHY por lo que predice como TUMOR un total de **0 muestras**.
  - Predicho HEALTHY (Clase 1): Estas son las **155 muestras** que realmente eran TUMOR pero el modelo las clasificó como HEALTHY (falso negativo).

- Segunda fila:
    - Predicho TUMOR (Clase 0): Hubo un total de **0 muestras** de HEALTHY cuando realmente TUMOR (falso positivo).
    - Predicho HEALTHY (Clase 1): Estas son las **1095 muestras** que realmente eran HEALTHY y el modelo las clasificó como HEALTHY (verdadero positivo).

Métricas Precision, Recall y F1

Estas métricas también podemos analizarlas en forma de tabla pero antes voy a explicar qué significa cada una de estas métricas.

```
              precision    recall  f1-score

  TUMOR            0.00      0.00      0.00
HEALTHY            0.88      1.00      0.93
```

Aplicado a lo proporcionado por el modelo obtenemos una tabla de esta forma:

	Precision	Recall	F1
TUMOR	0.0	0.0	0.0
HEALTHY	0.88	1.0	0.93

Analizando estos resultados podemos obtener las siguientes conclusiones:

- Primera fila:
    - **Precision**: No hay muestras predichas como TUMOR por lo que la precisión no se puede calcular.
    - **Recall**: Todas las muestras fueron clasificadas incorrectamente como HEALTHY, por lo que el modelo no está identificando ningún TUMOR.
    - **F1**: Al ser ambos 0, F1 es igual a 0.
- Segunda fila:
    - **Precision**: El 88% de las muestras clasificadas como HEALTHY eran realmente HEALTHY, lo podemos comprobar de la siguiente forma:

      $$\frac{Muestras\ clasificadas\ como\ HEALTHY}{Muestras\ totales} = \frac{1095}{1250} = 0,876 \simeq 0,88$$

    - **Recall**: El 100% de las imágenes que eran HEALTHY fueron clasificadas como HEALTHY, lo podemos comprobar de la siguiente forma:

      $$\frac{Muestras\ clasificadas\ como\ HEALTHY}{Muestras\ clasificadas\ como\ HEALTHY} = \frac{1095}{1095} = 1$$

    - **F1**: Es la combinación de ambas métricas y muestra un rendimiento de la siguiente forma:

      $$2 \times \frac{Precision \times Recall}{Precision + Recall} = 2 \times \frac{0,88 \times 1}{0,88 + 1} = 2 \times \frac{0,88}{1,88} \simeq 0,936$$

Accuracy

Por último contamos con Accuracy, es una métrica que se encarga de evaluar el **porcentaje de predicciones correctas realizadas** por un modelo en comparación con todas las predicciones realizadas. Es una medida muy utilizada ya que da una visión simplificada de qué tan bien funciona el clasificador que se está utilizando. Se calcula de la siguiente forma:

$$\frac{VP + VN}{VP + VN + FP + FN} = \frac{1095 + 0}{1095 + 0 + 0 + 155} = 0,876 \simeq 0,88 = 88\%$$

Siendo cada uno de estos parámetros:

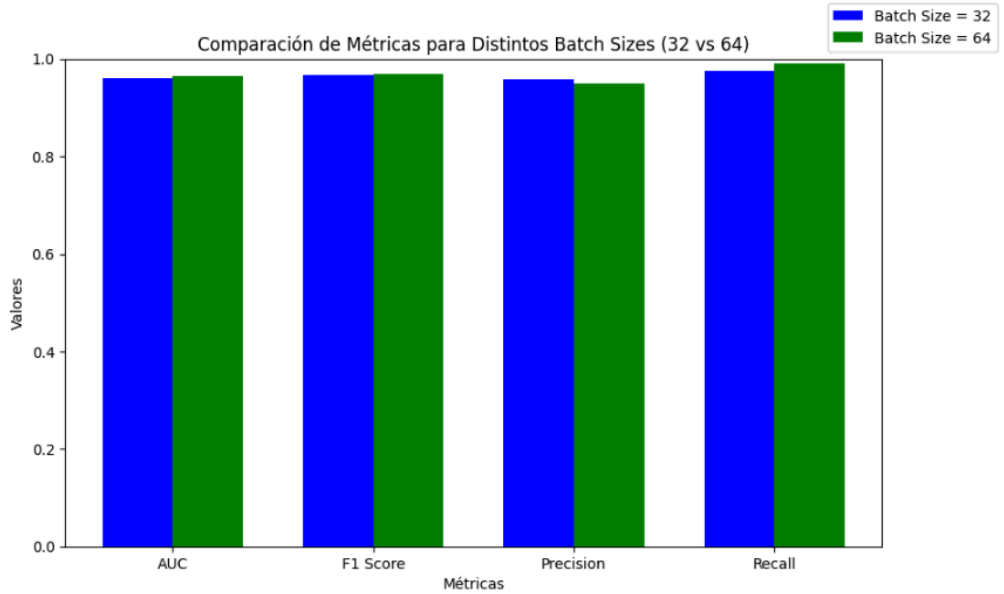
- VP: Verdaderos Positivos
- VN: Verdaderos Negativos
- FP: Falsos Positivos
- FN: Falsos Negativos

Así podemos concluir que el clasificador base clasifica el 88% de las imágenes como “correctas”, y sí, entre comillas ya que este valor está falseado, es decir, está fallando por completo en clasificar las imágenes de la clase TUMOR. En resumen, aunque este valor sea aparentemente alto tiene un problema significativo ya que existe un desequilibrio entre los datos y no está reconociendo ninguna muestra como TUMOR de manera correcta.

Análisis de parámetros en la red

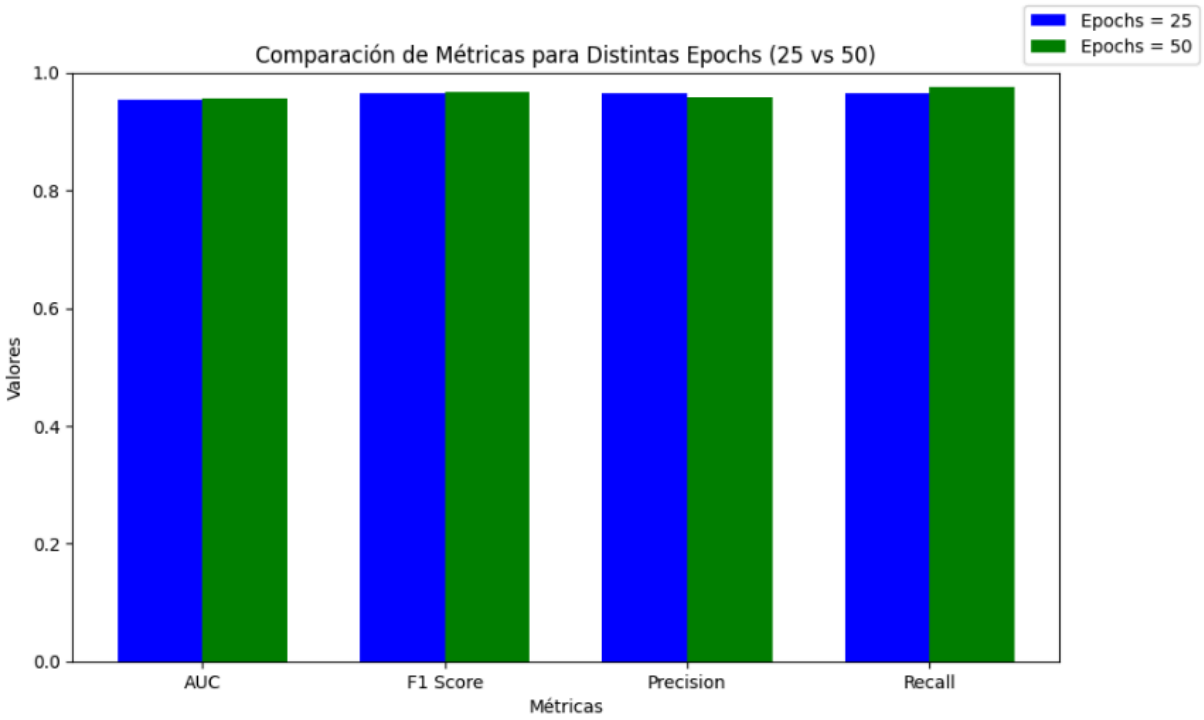
Hay multitud de parámetros para configurar una red neuronal convolucional como por ejemplo el tamaño de lote, números de épocas, número de clases por analizar (si es binaria o no), etc… Esto nos propone un primer problema a la hora de configurar los parámetros de nuestra red ya que habrá alguna combinación de parámetros que sea la mejor. Para ello vamos a modificar los siguiente parámetros:

- Tamaño de lote: Este parámetro se refiere al número de imágenes que se le pasan al modelo en cada iteración, por defecto en nuestra red base nos proponían utilizar un batch_size de 32, vamos a probar a utilizar batch_size de 64 para saber cuál tiene un mejor rendimiento:



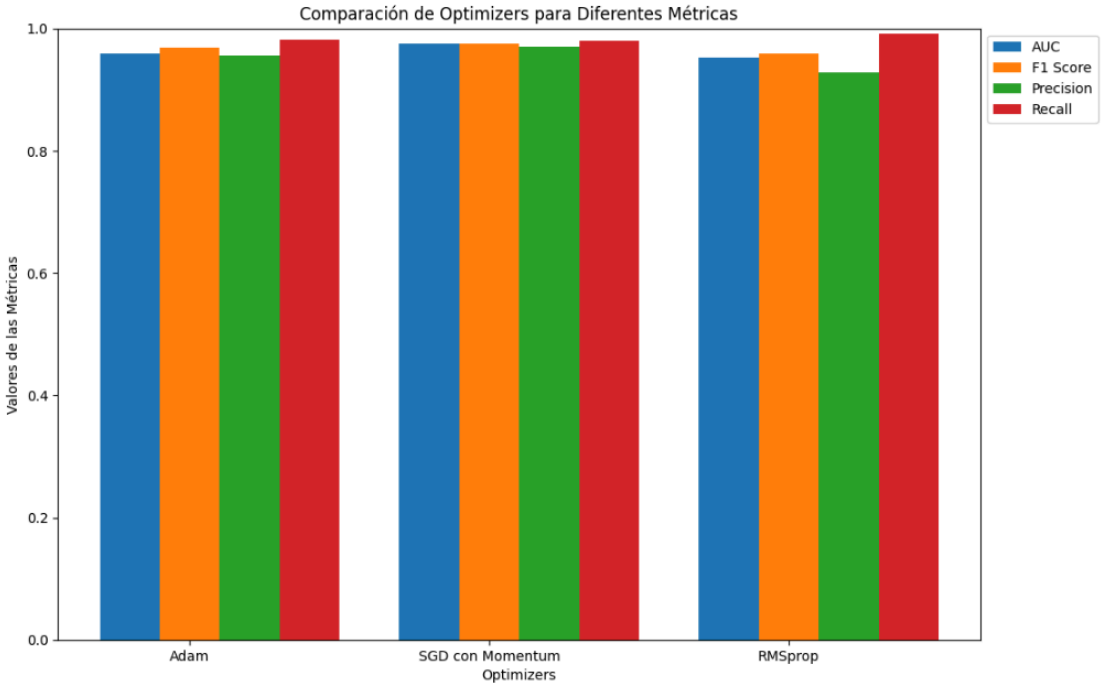
Como podemos observar no hay un gran cambio de un parámetro a otro por lo que trabajaremos con el batch_size 32 ya que ofrece una mejor exploración de mínimos locales.

- Número de épocas: Este parámetro representa el número de veces que todo el conjunto de datos pasa a través del modelo, vamos a hacer la prueba con 25 y con 50 épocas para ver si hay un incremento en su rendimiento.



Como podemos observar no hay un cambio significativo en las métricas por lo que usaremos un valor de epochs = 25 ya que con 50 estaríamos entrenando el modelo innecesariamente (se obtienen valores muy similares).

- Optimizers: Los optimizers son los algoritmos encargados de ajustar los pesos del modelo para minimizar la función de pérdida (loss). La elección de este parámetro puede tener un gran impacto en el rendimiento de la red. Las pruebas se realizarán con los siguientes optimizers:



Como podemos observar el optimizador SGD con Momentum parece ser el mejor optimizador según la gráfica, ya que presenta ligeras ventajas en AUC y F1 Score respecto a Adam y RMSprop.

Concluimos nuestros parámetros de la red con batch_size = 32, epochs = 25 y optimizer SGD with Momentum.

Básico

Clasificación binaria

Para poder realizar esta tarea es necesario hacer un ajuste a las etiquetas que tenemos en nuestro modelo, para ello guardamos en (X, y) los datos que vamos a recibir del DataSet.

```
1  X, y = load_data()
2
3  if nb_classes == 2:
4    y[y>0] = 1
```

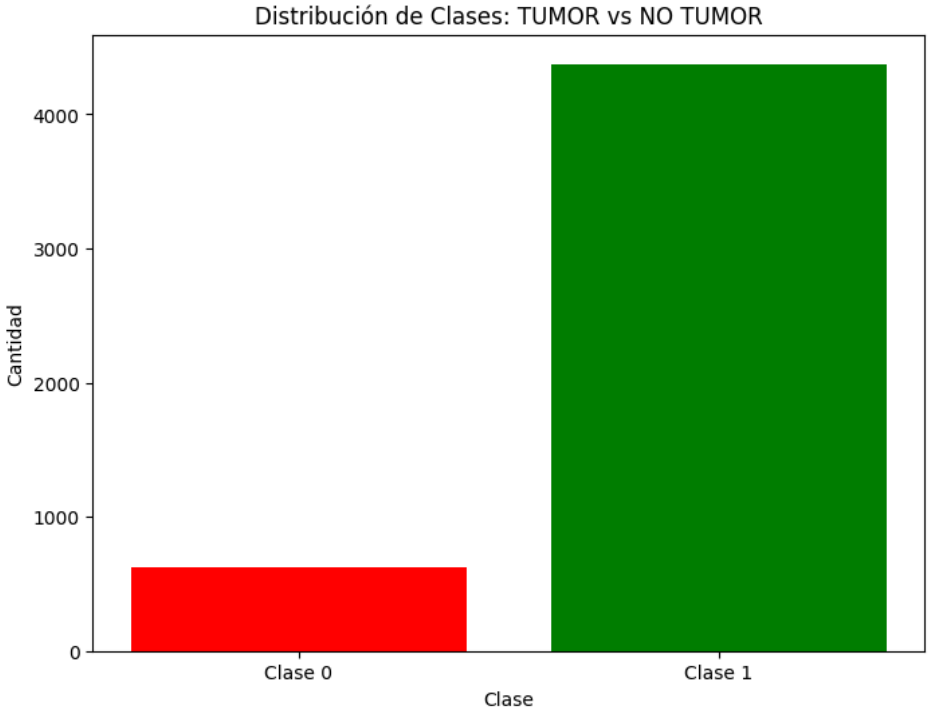
En X tendremos un tensor de la forma (n_samples, img_rows, img_cols, channels), su contenido será:

$$X \rightarrow (5000,\ 32,\ 32,\ 3)$$

Y en y tendremos una lista con las etiquetas que indican el tipo de tejido de cada imagen.

$$y \rightarrow [4,\ 5,\ 5,\ 0,\ 6,\ 6,\ 6,\ 4,\ \ldots]$$

Al aplicar y[y > 0] = 1 aquí ajustamos que cualquier etiqueta mayor de 0 será clasificada como NO TUMOR y la etiqueta 0 será clasificada como TUMOR, esto produce un desequilibrio de los datos ya que hay concentradas 7 clases en clase 1 frente a 1 en 0. Esto es un ejemplo de cómo se vería el reparto de clases en este DataSet con clasificación binaria aplicada a nuestro problema.



Aquí se muestra el desequilibrio existente entre las clases, como se puede observar en la imagen hay muchas más imágenes de clase 1 que de clase 0.

Métrica AUC y F1

En este apartado se analizarán las métricas obtenidas del modelo de clasificación binaria (TUMOR vs HEALTHY) entrenado y evaluado en el conjunto de datos. El código para entrenar nuestro modelo se ejecuta de la siguiente manera:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=24)
model = create_model()
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size)
y_pred = model.predict(X_test)

auc = roc_auc_score(y_test, y_pred)
print(f"Resultado de la métrica AUC: {auc:.5f}")
f1 = f1_score(y_test, np.round(y_pred))
print(f"Resultado de la métrica F1: {f1:.5f}")
```

El valor de AUC obtenido fue un 0.97467, esto indica la capacidad del modelo para discriminar entre las clases TUMOR y HEALTHY.

```
Resultado de la métrica AUC: 0.97467
```

En cuanto a la métrica F1 el valor obtenido fue un 0.97589, esto indica un buen balance entre la capacidad del modelo para detectar verdaderos positivos y minimizar los falsos positivos.

```
Resultado de la métrica F1: 0.97589
```

F1 es la media armónica entre precision y recall, esto hace que sea una métrica muy útil en casos donde hay un claro desbalance entre clases, como es nuestro caso. Los valores de estas métricas podemos obtenerlas haciendo uso del método `classification_report` de la clase sklearn.metrics

```
target_names = ['TUMOR', 'HEALTHY'] if nb_classes ==  2 else ['TUMOR','STROMA','COMPLEX','LYMPHO','DEBRIS','MUCOSA','ADIPOSE','EMPTY']
print(classification_report(y_test, y_pred_redondeado, target_names=target_names))
```

```
              precision    recall  f1-score

       TUMOR       0.85      0.82      0.84
     HEALTHY       0.97      0.98      0.98
```

Con toda esta información no queda nada claro qué quiere decir cada cosa, para ello hacemos uso de la matriz de confusión, la matriz de confusión nos dice cuántos falsos positivos y verdaderos negativos tenemos, es decir, información errónea que ha clasificado nuestro modelo.

```
y_pred_redondeado = np.round(y_pred)
matriz_confusion = confusion_matrix(y_test, y_pred_redondeado)
print(f"Matriz de Confusión:\n{matriz_confusion}\n")
```

```
Matriz de Confusión:
[[108  23]
 [ 19 850]]
```

Esto quiere decir que ha clasificado **23 falsos positivos**, es decir, 23 imágenes que son NO TUMOR fueron clasificadas como TUMOR. Además también clasifica **19 falsos negativos**, es decir, 19 casos de imágenes que son TUMOR y fueron clasificados como NO TUMOR. El resto de valores son información correctamente clasificada, **108 imágenes** fueron clasificadas como **TUMOR** cuando son TUMOR y **850 imágenes** como **NO TUMOR** cuando realmente son NO TUMOR.

En resumen, obtenemos un valor alto de AUC y de F1 pero hay que tener en cuenta que en la matriz de confusión aparecen falsos negativos y falsos positivos que habrá que corregir.

Aplicación de la validación cruzada

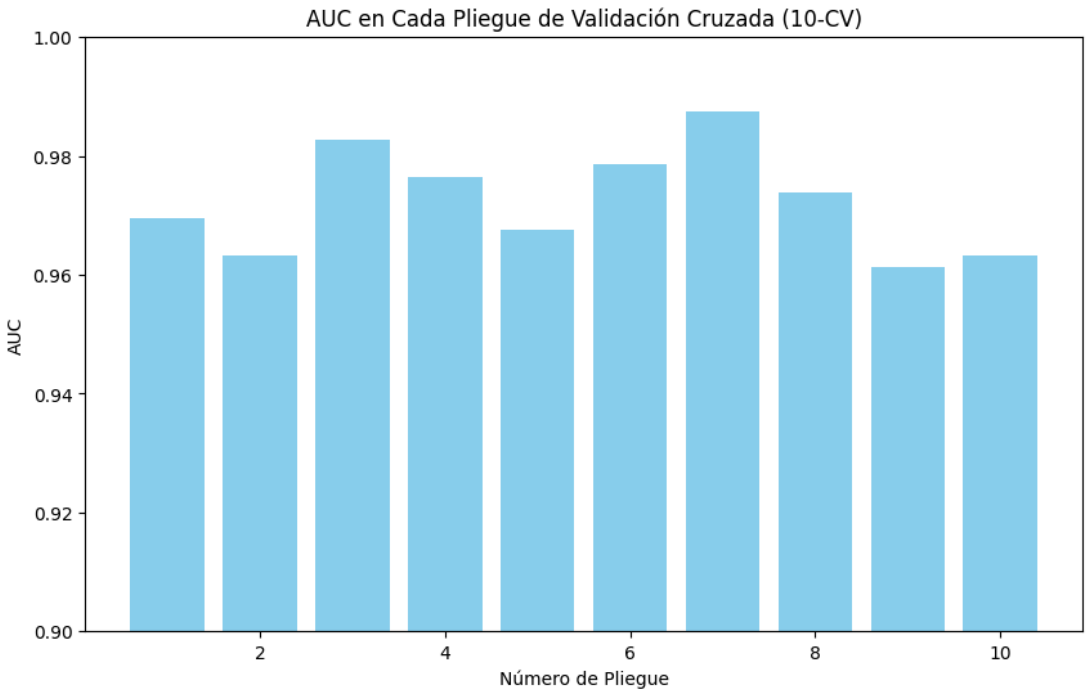
Vamos a hacer uso de la validación cruzada (o cross-validation), más en concreto del 10-CV. La validación cruzada se utiliza para evaluar la capacidad de generalización del modelo sobre datos no vistos. La validación cruzada hace uso de todo el conjunto de datos en lugar de solo un % para entrenamiento. El objetivo de este método es asegurar que el modelo que tenemos no esté sobre ajustado a los valores de entrenamiento (overfitting), es decir, aprende demasiado de los datos que ya tiene existentes y no es tan susceptible a aprender de los nuevos datos que pueda tener. Vamos a utilizar el 10-CV. Para ello se utilizará la clase **StratifiedKFold** para mantener la proporción de clases en cada pliegue ya que hay un desequilibrio presente en los datos.

```
skf = StratifiedKFold(n_splits=10) # dividimos el dataset en 10 pliegues
auc_scores_10cv = []   # guardamos el area bajo la curva en cada pliegue

for train_index, test_index in skf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]   # dividimos datos e
    y_train, y_test = y[train_index], y[test_index]

    model = None
    model = create_model()  # creamos un nuevo modelo en cada pliegue
    model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,\
              verbose=0, validation_data=(X_test, y_test))  # entrenamos

    y_pred = model.predict(X_test).ravel()       # realiza predicciones
    auc_10cv = roc_auc_score(y_test, y_pred)     # area bajo la curva roc
    auc_scores_10cv.append(auc_10cv)
```

Se obtuvo de media un valor de AUC de 0.9724 con una desviación estándar de 0.084.

```
Media de AUC (10-CV): 0.9724 en los 10 pliegues
Desviación estándar de AUC (10-CV): 0.0084 en los 10 pliegues
```

Esto nos indica que el modelo tiene un rendimiento consistente entre los pliegues y no varía en gran cantidad, es decir, nuestro modelo tiene buena capacidad de generalización.



La gráfica del AUC por cada pliegue de 10-CV muestra un rendimiento consistente del modelo con valores que oscilan entre 0.96 y 0.98. Aunque hay variaciones la mayoría de valores se mantienen cercanos a la media, lo cual indica que el modelo tiene una buena capacidad de generalización y un rendimiento estable.

Estudio estadístico de Wilcoxon

Para realizar el estudio estadístico de Wilcoxon he recopilado distintos optimizadores para comparar cuál de ellos es el que mejor valor de AUC en promedio nos da en nuestro modelo. Cada optimizador utilizará un modelo distinto ya que habrá que adaptar el optimizador de esa red. El optimizador será el encargado de ajustar los pesos y minimizar nuestra función de pérdida para maximizar nuestro resultado en la medida de lo posible. Este estudio al ser estadístico nos proporcionará un valor entre 0 y 1 que deberemos saber interpretar.

La primera comparación fue entre ‘Adam’ y ‘RMSprop’, inicialmente Adam nos proporcionaba valores de AUC bastantes bajos con respecto a RMSprop pero se estabilizó y acabó teniendo muy buenos resultados superando a RMSprop, el valor de p como resultado del Test de Wilcoxon que resultó fue:

```
Wilcoxon test entre adam y rmsprop: p-value = 0.695
```

Esto indica que en términos estadísticos quiere decir que existe un 69.50% de probabilidad de que las diferencias entre ambos sean al azar, es decir, no se puede concluir que uno de los modelos sea mejor que el otro por lo que se rechaza esta hipótesis. En resumen, ambos modelos tienen un rendimiento similar para el problema que estamos abordando.

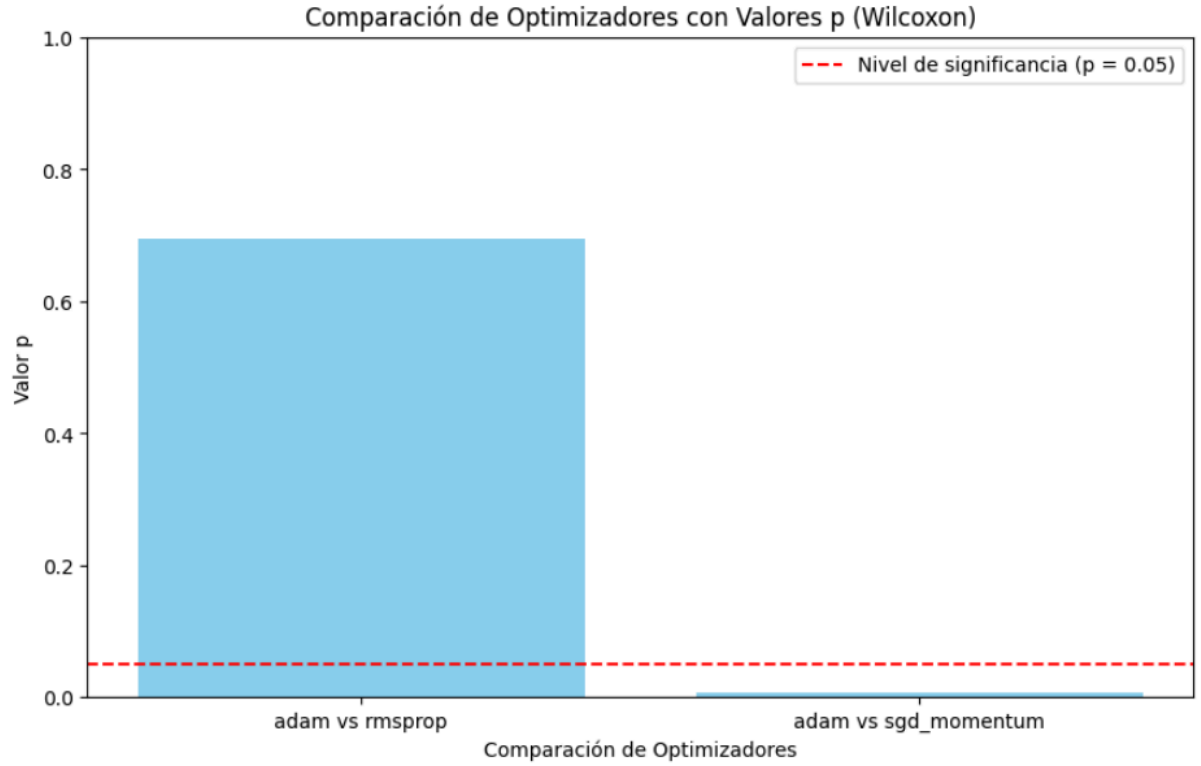
La siguiente comparación fue entre ‘Adam’ y ‘SGD con momentum’, para aclarar, SGD con momentum es un optimizador basado en descenso por gradiente estocástico (SGD). Añadiendo momentum 0.9 a este optimizador indicamos que el 90% del gradiente anterior

se utilizará para influir en la actualización actual frente al otro 10% que es el gradiente calculado en la actualización actual. El resultado del Test de Wilcoxon obtenido frente a Adam fue:

```
Wilcoxon test entre adam y sgd_momentum: p-value = 0.006
```

Como podemos observar aquí si obtenemos un valor de p menor que 0.05, una vez encontramos un valor así podemos afirmar que existe una diferencia significativa entre estos dos modelos por lo tanto podemos decir que SGD con momentum 0.9 es mejor que Adam en este contexto. También hay que tener en cuenta que el valor de p es 0.006 por lo que hay una diferencia extremadamente fuerte entre ambos optimizadores.

La gráfica comparativa para hallar cuál es el mejor optimizador para nuestro caso es la siguiente:



Aquí se observa lo que comenté anteriormente, es decir, a partir de un valor de $p < 0.05$ podemos afirmar que existe una diferencia significativa entre los modelos pero nuestro valor de p es 0.006 osea que podemos afirmar que hay una diferencia bastante alta entre los optimizadores Adam y SGC con Momentum 0.9.

Aumento de datos con ImageDataGenerator

Cómo ya hemos visto existe un desequilibrio de muestras entre las clases que tenemos en nuestra clasificación binaria. Hay un total de 4375 imágenes de clase 1 (NO TUMOR) y un total de 625 imágenes de clase 0 (TUMOR). Para abordar esta tarea necesitamos utilizar la clase ImageData Generator para generar más muestras de las muestras que ya tenemos pertenecientes a la clase 0 (TUMOR). A estas imágenes le aplicaremos filtros para generar las nuevas imágenes, estos son los filtros utilizados:

```python
generador_datos = ImageDataGenerator(
    rotation_range=15,                  # rotacion de (-15, +15) grados
    width_shift_range=0.15,             # desplazamiento horizontal de hasta 15%
    height_shift_range=0.15,            # desplazamiento vertical de hasta 15%
    horizontal_flip=True,               # voltear imagenes en horizontal
    fill_mode='reflect',                # rellenar pixeles con el reflejo de los cercanos
    shear_range=0.2,                    # distorsion del 20%
    zoom_range=[0.8, 1.2],              # zoom entre 80% y 120%
)
```

Inicialmente utilizaremos estos ajustes, vamos a hacer más pruebas con otros filtros para ver cuál de todos nos ofrece un mejor resultado.

Con estos filtros se generarán las nuevas imágenes, como comparación de la imagen real y la imagen nueva modificada de la original se generan las siguientes imágenes:

Imagen real

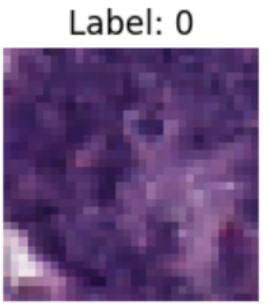


Imagen generada

Una vez ajustados los filtros generamos un total de 3750 imágenes de clase 0 para así obtener una muestra equilibrada. A esta nueva muestra vamos a aplicarle 10-CV para obtener valores de AUC sobre el rendimiento de nuestro modelo.

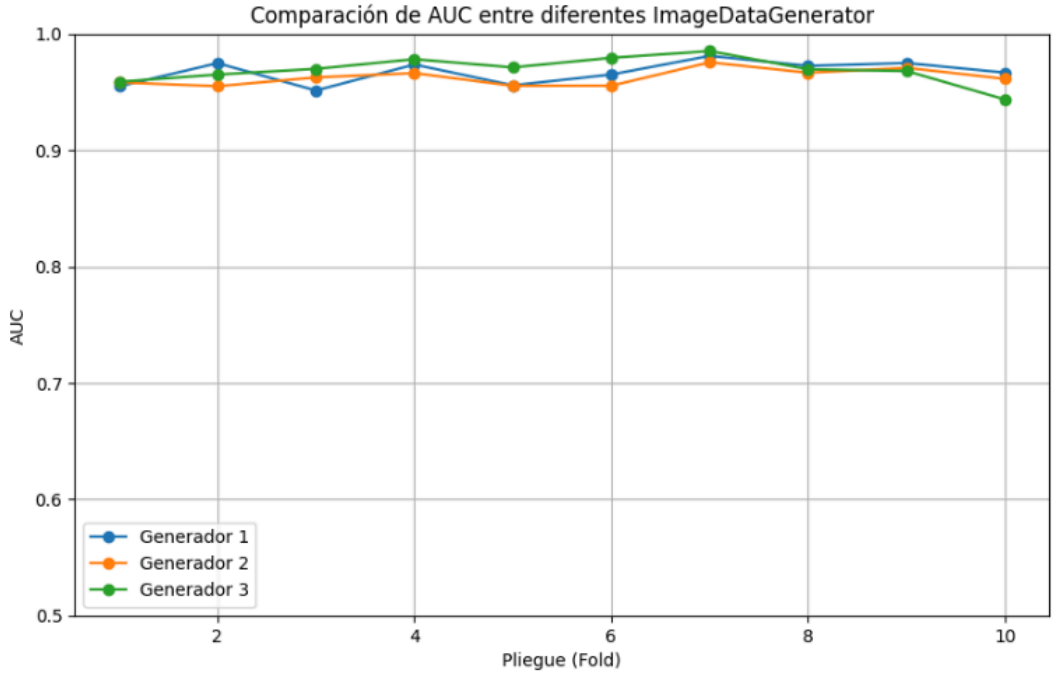
Los generadores de datos que han sido utilizados han sido los siguientes:

```python
generador_1 = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    fill_mode='reflect',
    shear_range=0.1,
    zoom_range=[0.9, 1.1]
)
```

```python
generador_3 = ImageDataGenerator(
    rotation_range=5,
    width_shift_range=0.05,
    height_shift_range=0.05,
    horizontal_flip=False,
    fill_mode='constant',
    shear_range=0,
    zoom_range=[0.95, 1.05]
)
```

```python
generador_2 = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest',
    shear_range=0.15,
    zoom_range=[0.8, 1.2]
)
```

Al realizar varias pruebas con diferentes valores de los filtros sobre el constructor generador de datos obtenemos valores de AUC que se asemejan bastante a los obtenidos utilizando 10-CV sin aumento de datos. La gráfica donde se muestran los resultados es la siguiente:



El Generador 1 es la mejor configuración, ya que logra los valores de AUC más altos y consistentes entre los pliegues, mostrando un buen equilibrio entre todos los pliegues. Las transformaciones más agresivas del Generador 2 generan algo de variabilidad, mientras que el Generador 3, con transformaciones mínimas, es el menos efectivo.

Intermedio

Clasificación multiclase

El objetivo de esta tarea es clasificar todas las categorías que tenemos disponibles en el DataSet. Para ello definimos un nuevo modelo para una red neuronal convolucional (CNN), el nuevo modelo consta de 2 capas convolucionales y 2 capas densas, el modelo ha sido definido de la siguiente forma:

```
model = Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(img_rows, img_cols, 3)),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.5),                              # regularización
    layers.Dense(nb_classes, activation='softmax')    # 8 neuronas de salida
])
model.compile(optimizer=optimizer_usado, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Se utiliza Dropout(0.5) para evitar el ‘overfitting’, además se ha ajustado como función de pérdida ‘sparse_categorical_crossentropy’ para facilitar el preprocesamiento y simplificar el trabajo con etiquetas enteras para nuestro problema de clasificación multiclase.

```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=24)

model = create_model_multiclass()
history = model.fit(X_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(X_val, y_val))

y_pred_multiclass = model.predict(X_val)

y_val_one_hot = tf.keras.utils.to_categorical(y_val, num_classes=nb_classes)    # y_val pasa a una matriz binaria

auc_score = roc_auc_score(y_val_one_hot, y_pred_multiclass, average='macro', multi_class='ovr')  # calculo de AUC

print(f'AUC (macro promedio): {auc_score:.4f}')
```

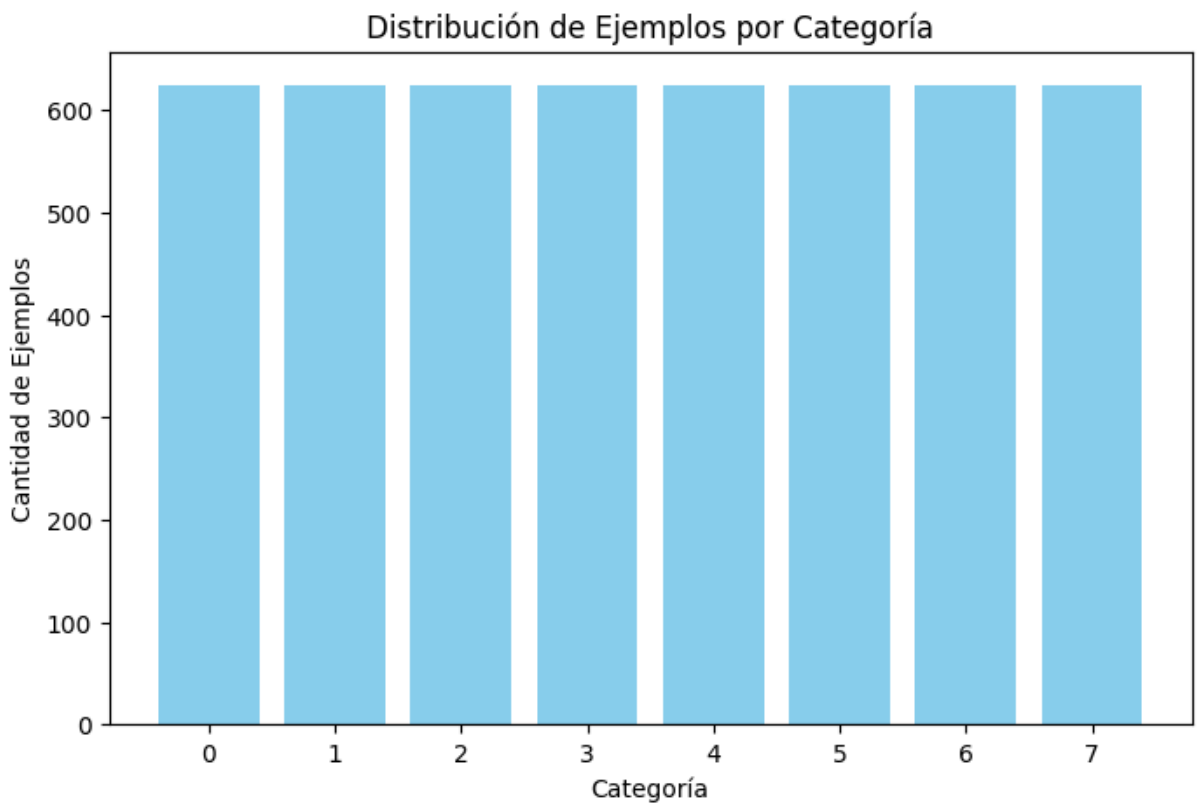
Al entrenar nuestro modelo obtenemos un valor de AUC:

```
AUC (macro promedio): 0.9778
```

Este valor de AUC nos indica que el modelo es efectivo discriminando entre las distintas categorías existentes en nuestro problema.

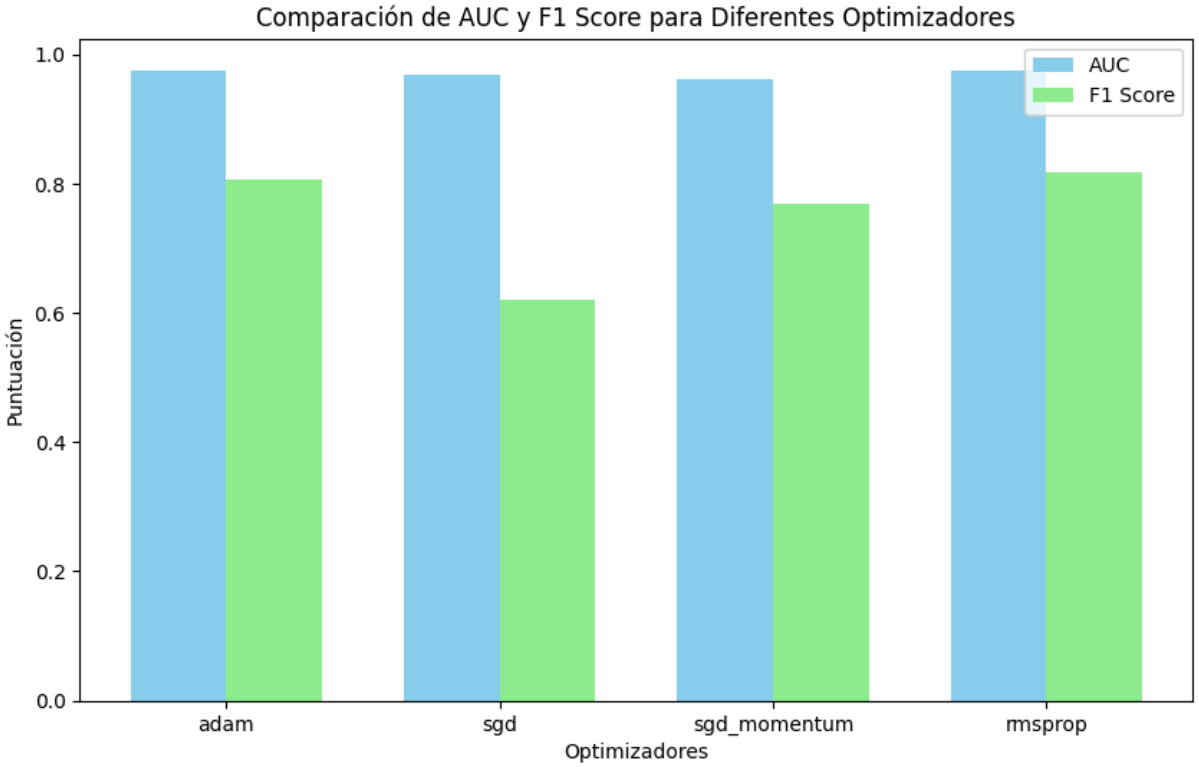
Estudio de métricas AUC y F1

Ahora tenemos un equilibrio de clases entre todas las categorías, una gráfica que demuestra esto es la siguiente:



Como ahora sí existe este equilibrio entre todas las clases podemos utilizar los distintos optimizadores para evaluar las métricas AUC y F1 para saber cuál es el mejor adaptado a

nuestro caso específico. Para este caso vamos a comparar Adam, SGD, SGD con Momentum 0.9 y por último RMSprop. Al ejecutar cada uno de estos modelos y entrenarlo obtenemos los siguientes resultados:



Podemos observar que Adam y RMSprop nos proporcionan un valor similar de ambas métricas, ya que en el anterior caso utilizamos SGD con Momentum 0.9 en este vamos a utilizar Adam para llevar al modelo a una convergencia más rápida.

Ajustes combinados

Para hacer esta tarea vamos a modificar los distintos parámetros de convolución como lo son las capas de nuestro modelo y además la cantidad de filtros en nuestro modelo.

Por ello debemos cambiar la manera en la que cargamos las imágenes del dataset de la siguiente manera:

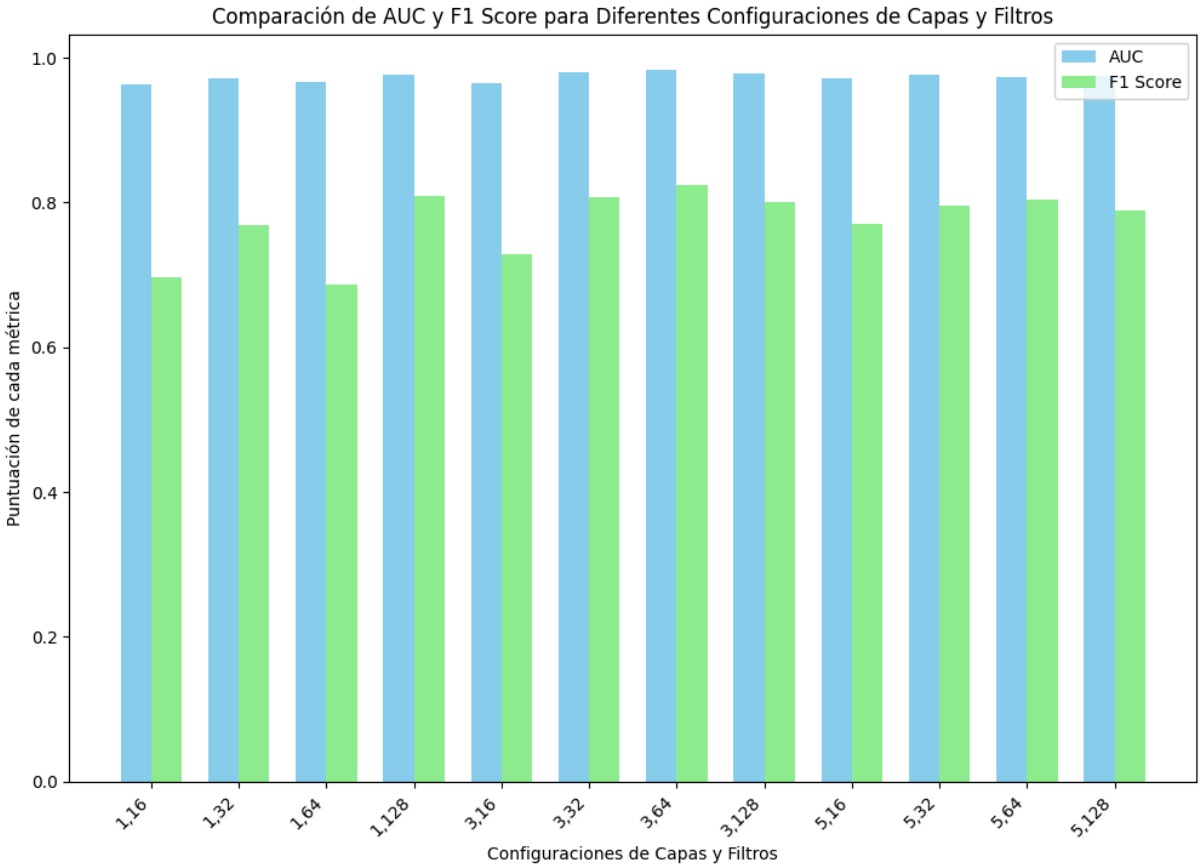
```
def load_data_resized(name="colorectal_histology"):          # misma manera de cargar los datos pero reescaland
    train_ds = tfds.load(name, split=tfds.Split.TRAIN, batch_size=-1)
    train_ds['image'] = tf.map_fn(lambda x: tf.cast(x, tf.float32) / 255.0, train_ds['image'], dtype=tf.float32)
    train_ds['image'] = tf.image.resize(train_ds['image'], (16, 16))    # 16x16 pixeles cada imagen
    numpy_ds = tfds.as_numpy(train_ds)
    X, y = numpy_ds['image'], numpy_ds['label']
    return np.array(X), np.array(y)

X, y = load_data_resized()
```

Indicando así que las imágenes se van a cargar de 16x16 píxeles. Las imágenes cargadas son las siguientes:

Por otro lado también se modificará el tamaño de imagen ya que van a ser reescaladas a 16x16 píxeles. Para lograr entrenar el modelo con varias pruebas de capas y filtros modificamos la función que nos crea el modelo para iterar sobre listas con los parámetros que se ajusten a lo que necesitamos, haremos la prueba con modelos de 1, 3 y 5 capas y con 16, 32, 64 y hasta 128 filtros por modelo. El modelo resultante sería algo por este estilo:

```python
def create_model_n_layers_n_filters(n_layers, n_filters):
    model = tf.keras.Sequential()

    for i in range(n_layers):     #  añadir n_layers capas al modelo
        if i == 0:
            model.add(tf.keras.layers.Conv2D(n_filters, (3, 3), activation='relu', input_shape=(16, 16, 3)))
        else:
            model.add(tf.keras.layers.Conv2D(n_filters, (3, 3), activation='relu'))

        if i > 0 and i % 3 == 0:
            # Añadir una capa de MaxPooling con tamaño reducido
            model.add(tf.keras.layers.MaxPooling2D((2, 2), strides=(2, 2), padding='same'))

    model.add(tf.keras.layers.GlobalAveragePooling2D())     # evitamos flatten y usamos globalaveragepooling2

    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(128, activation='relu'))
    model.add(tf.keras.layers.Dense(64, activation='relu'))
    model.add(tf.keras.layers.Dropout(0.5))
    model.add(tf.keras.layers.Dense(8, activation='softmax'))   # por ser multiclase se usa softmax

    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    return model
```

Como se puede observar vamos añadiendo tantas capas (n_layers) cómo contenga nuestra variable con la que llamamos a la función. Durante el entrenamiento obtenemos valores de AUC y F1 con cada uno de los parámetros con reescalado a 16x16 píxeles. La gráfica resultante es la siguiente:



Como podemos observar los valores de AUC y F1 son bastante similares en cada uno de los distintos parámetros, de esto podemos sacar una conclusión ya que a mayor número de capas de filtros podemos observar que no cambia significativamente los valores de AUC y F1 por lo que estaríamos sobreentrenando el modelo para no obtener cambios. En cuanto a la comparación con las imágenes de 32x32 píxeles se nota un ligero incremento ya que ninguno de estos valores de AUC ronda el 0.95 de AUC mientras que con 32x32 había bastantes casos con ese valor ya que era el promedio.

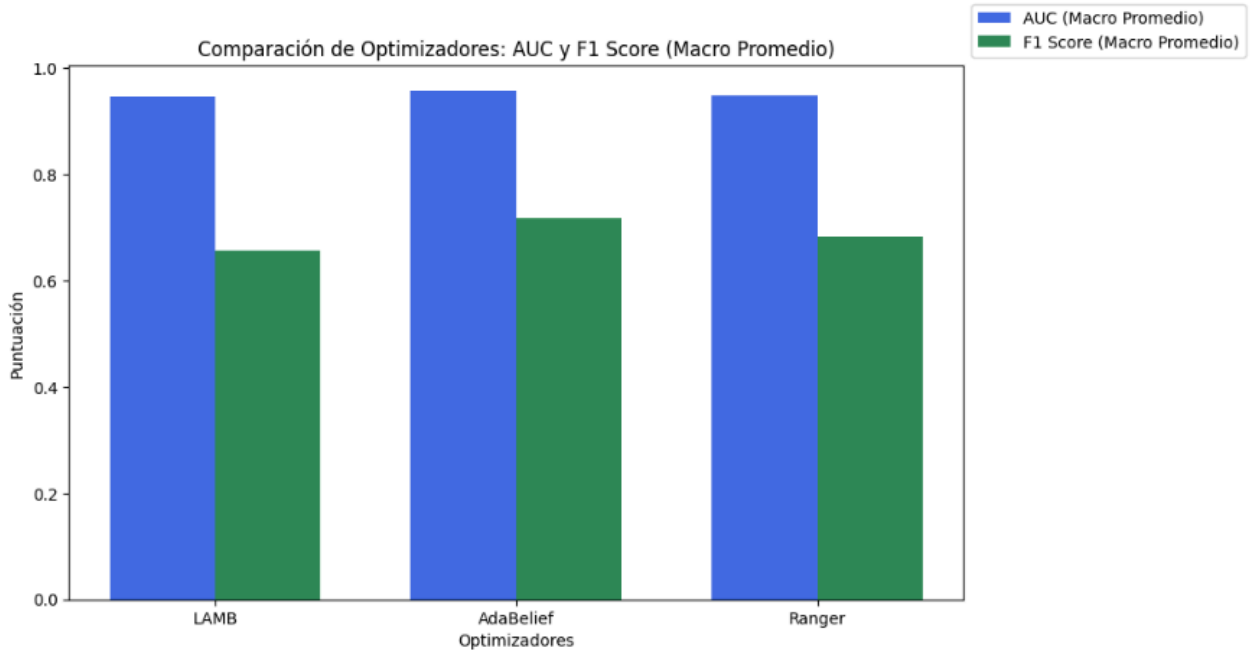
Ajuste de algoritmo de optimización

En un inicio se pretendía analizar los distintos optimizadores que están disponibles fuera de las librería de tensorflow.keras, se ha analizado el optimizador Ranger que es una mezcla entre RAdam y Lookahead, RAdam es utilizado para la convergencia inicial del modelo y Lookahead evita que quede atrapado en mínimos locales. Se analizó el rendimiento del modelo de la siguiente manera:

```
AUC (macro promedio usando Ranger): 0.9579
F1 Score (macro promedio usando Ranger): 0.7130
```

Inicialmente se probó Ranger con 10 épocas pero los valores de AUC rondaban los valores entre 0.93 y 0.94 con varios decimales de por medio, aumentando este valor de épocas a 30 se incrementó a 0.95 y ahí se quedó por muchas épocas que añadiese. Para poder utilizar este optimizador hay que importar tensorflow-addons que es la librería la que tiene este optimizador. Además esta librería también incluye el resto de optimizadores que vamos

a usar que son LAMB y AdaBelief. Al entrenar los modelos con los distintos optimizadores se obtuvieron distintos valores de AUC y F1, para poder compararlos podemos observar la siguiente gráfica:



Aquí podemos ver todos los valores obtenidos de AUC y F1 de los distintos optimizadores utilizados, como conclusión podemos decir que ambos se parecen bastante y no hay uno en específico que destaque sobre el resto.

Test estadísticos

Ahora vamos a analizar los distintos valores de p que nos proporciona el test de Wilcoxon, para ello hemos aplicado la validación cruzada (10-Cross Validation) para cada uno de los optimizadores que analizamos anteriormente para así poder averiguar cuál es el que mejor resultado nos va a ofrecer aplicado a nuestro problema. Como ya observamos en la imagen anterior todos los optimizadores dan un rendimiento similar, una vez se le aplicó 10CV estos fueron los valores de AUC y F1 que se obtuvieron:

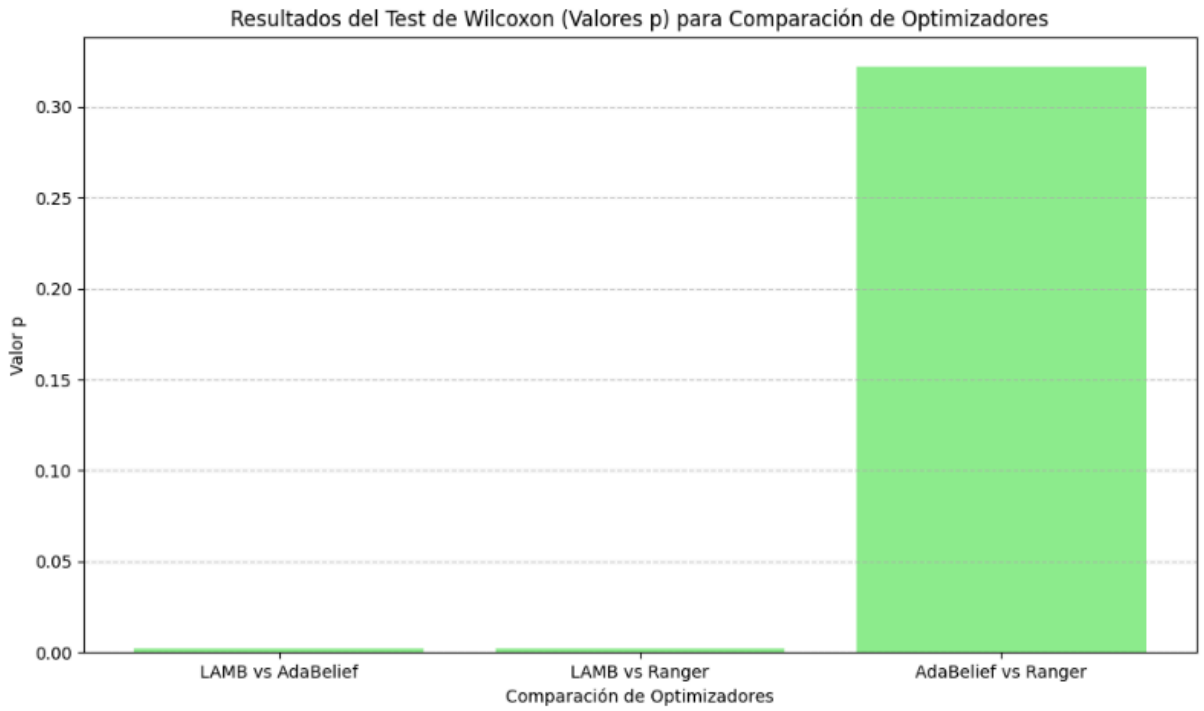
```
Optimización con LAMB completada. AUC medio: 0.9432, F1 Score medio: 0.6289
```

```
Optimización con AdaBelief completada. AUC medio: 0.9579, F1 Score medio: 0.6525
```

```
Optimización con Ranger completada. AUC medio: 0.9568, F1 Score medio: 0.6975
```

Siendo aparentemente el mejor Ranger debido a su poca diferencia entre el valor de AUC frente a AdaBelief pero su más notable diferencia en el valor de F1.

Una vez ha sido calculado cada uno de los p_values del test de Wilcoxon de cada uno de estos optimizadores por parejas, es decir, LAMB vs AdaBelief, LAMB vs Ranger, y así sucesivamente se obtuvo la siguiente gráfica:



Como podemos observar la gráfica nos indica que el optimizador LAMB tiene un comportamiento totalmente diferente a los otros dos optimizadores que son Ranger y AdaBelief. Estos optimizadores tienen un comportamiento similar y además sus valores de AUC y F1 son relativamente similares por lo que podemos concluir que no existe una diferencia significativa entre estos dos optimizadores al contrario que pasa con LAMB.

Avanzado

Redes pre-entrenadas con fine-tunning

Para realizar este apartado se ha utilizado la red pre-entrenada de ResNet50, esta red está entrenada con el conjunto de datos 'imagenet', es decir, ya "conoce" características de las imágenes como bordes, texturas, etc. Con este fenómeno podríamos decir que puede transferir el conocimiento aprendido sobre unas imágenes a otras reduciendo así el tiempo de ejecución usando estas redes ya que vienen con pesos preestablecidos. Para poder adaptar esta red a nuestro problema se hace uso del fine-tunning que hace referencia a la capacidad que tiene el modelo para aprender en sus últimas capas con el dataset que se le indique, en nuestro caso, colorectal_histology. Este modelo ha sido entrenado con 50 épocas en las que hemos obtenido un valor de AUC promedio del 0.78.

```
AUC (Area Under the Curve) para el conjunto de validación: 0.7899
```

Esto podría deberse a que al tener tan pocas imágenes el modelo tenga esa carencia de imágenes y esté viendo siempre las mismas imágenes de las que aprender. Por otro lado cabe destacar que el modelo a medida que se le incrementa el número de épocas por las que pasa el modelo crece su valor de AUC y disminuye su pérdida aunque lo hace de poco a poco.

Capítulo II : Detección de incoherencias en una evaluación por pares

Objetivo

El principal objetivo de este desafío es implementar soluciones a problemas en el ámbito del procesamiento de lenguaje natural o (NLP) usando redes neuronales.

Básico

Preprocesado básico de palabras

La primera tarea a realizar es preprocesado básico de palabras para ello debemos seguir unos pasos que serán cruciales para preprocesar el dataset que tenemos:

1) Conversión de mayúsculas a minúsculas:

   Se deben de transformar todos los textos de mayúsculas en minúsculas para no causar ambigüedad por la capitalización de las palabras, es decir, un ejemplo concreto sería que las palabras 'Casa' y 'casa' deben ser tratadas de la misma forma.

2) Tokenización:

   Uso de un tokenizador para dividir las frases en palabras, el uso del mismo además sirve para evitarnos incluir signos de puntuación que nos dificultan el análisis del texto quedándonos exclusivamente con caracteres alfanuméricos.

3) Stemming:

   Con esta técnica reducimos las palabras que anteriormente dividimos con el tokenizador convirtiendo y analizando cada una de las palabras que conforman las frases en su raíz. Esto es completamente necesario ya que el español tiene una gran cantidad de sufijos verbales, de género y número que causan redundancia.

4) Eliminación de 'stopwords':

   Para evitar el análisis de palabras que no aportan nada o son poco relevantes denominadas stopwords. Estas palabras son por ejemplo 'de', 'el', 'la', 'en', generalmente son determinantes artículos, conjunciones, preposiciones, verbos auxiliares como ser, estar, haber o tener entre muchos otros, pronombres, interjecciones, adverbios de lugar/tiempo como 'ahí', 'aquí', 'mañana', etc.

Preparo un pequeño ejemplo para comprobar el contenido de las stopwords presente en nltk. El ejemplo es el siguiente:

```
nltk.download('stopwords')  # descargar stopwords
stopwords_es = set(stopwords.words('spanish'))
print(stopwords_es)
```

La salida proporcionada es esta, aquí podemos comprobar cuáles son algunas de las palabras que están contenidas en las stopwords de nltk.

```
{'estarás', 'sentidas', 'era', 'de', 'sin', 'que', 'sois', 'al', 'hubieron', 'como', 'tengan',
```

Ahora vamos a aplicar todo esto mencionado anteriormente para ver cómo procesaría las distintas frases de ejemplo. Las frases utilizadas son las siguientes:

```
frases = [
    "Estoy estudiando cómo funcionan las redes de procesamiento de lenguaje natural.",
    "Desafíos de programación es la mejor asignatura, ¿verdad?"
]
```

Pues siguiendo los pasos que se mencionaron anteriormente debemos de tokenizar cada una de las frases, aplicarle stemmer para convertirlas a su raíz y eliminar las stopwords que estén presentes en el texto. Este es el resultado obtenido:

```
Original: Estoy estudiando cómo funcionan las redes de procesamiento de lenguaje natural.
Procesada: estudi com funcion red proces lenguaj natural
--------------------------------------------------
Original: Desafíos de programación es la mejor asignatura, ¿verdad?
Procesada: desafi program mejor asignatur verd
```

Aquí observamos cada una de las palabras transformada a su palabra raíz (stemming) con eliminación de preposiciones entre muchas otras (stopwords) y además eliminación de signos de puntuación (tokenización). Una vez comprendido todo este proceso se procede a realizar el proceso frente al dataset del que disponemos.

El proceso es exactamente el mismo que el que acabo de hacer para mostrar las frases que eran de prueba solo que ahora con el dataset con una única condición ya que si ese feedback está vacío no se llega a modificar nada, voy a explicarlo con la siguiente tabla:

feedback	feedback prep
No ha puesto el símbolo de CC	puest simbol cc
Este trabajo posee introducción, en la cual se indica el tema. Muy bien presentado.	trabaj pose introduccion indic tem bien present

Esto es un ejemplo real del dataset después de haber preprocesado el feedback para cada una de las entradas, en caso de estar vacío previamente no se modifica y se queda en blanco.

Aplicación de la validación cruzada 10-CV

Para este apartado necesitamos realizar una tarea previa, la creación de un modelo regresor, este modelo tiene como objetivo predecir un valor numérico asociado a un texto de un rango entre 0 a 3. Para ello va a ser construido con Keras como en el anterior desafío. El modelo inicial consta de una capa de entrada, dos capas densas y una capa de salida lineal para hacer predicciones continuas. Un requisito adicional que tiene el modelo que he

diseñado es que necesita que los datos de entrada estén previamente vectorizados, es decir, el texto vectorizado.

¿Qué es la vectorización de textos?

La vectorización de textos, en nuestro caso **Bag of Words** (BoW), representa todo el texto de entrada como una bolsa de palabras ignorando el orden teniendo cada palabra de esta bolsa un valor de 1 que contará más al vector de palabras en función del número de veces que aparece en el texto, pongo un ejemplo práctico.

Para la frase “El gato come y el gato maúlla” se genera el siguiente vocabulario ordenado por orden alfabético:

‘come’	‘el’	‘gato’	‘maúlla’	‘y’

Se vectoriza este vocabulario en función del número de veces que aparece esa palabra en el texto, quedaría de esta forma:

1	2	2	1	1

Esta es la representación de la matriz BoW (1x5) de la frase “El gato come y el gato maúlla”.

Creación del modelo regresor

Este es el código asociado a la creación del modelo regresor inicial que será utilizado para realizar las predicciones

```
##########################################
#
#     Creación del modelo regresor
#
##########################################

def create_regressor_model(X, y): # X sin transformar
    model = Sequential([
        Dense(128, activation='relu', input_shape=(X_vectorizado.shape[1],)),    # capa de entrada
        Dropout(0.2),
        Dense(64, activation='relu'),
        Dense(1, activation='linear')     # capa de salida para prediccion continua
    ])

    return model
```

Como anteriormente mencioné tiene una capa de entrada con la forma del tamaño de columnas que tiene la matriz de palabras (BoW) que tenemos presente en los datos, es decir, representa el número de palabras en el vocabulario presente en ese texto.
Cuenta además con una capa de ajuste, una capa densa y una capa de salida con activación ‘linear’ para predecir valores continuos.

Una vez establecido el modelo podemos empezar a aplicar 10-CV a nuestro modelo. La manera de aplicarlo es exactamente igual que en el análisis de imágenes pero con un par de características diferentes. Una de ellas es la necesidad de vectorizar previamente el

texto por lo que hay que aplicar un vectorizador, en mi caso, al usar BoW he utilizado CountVectorizer.

```
vectorizador = CountVectorizer(max_features=500)  # BoW con las 500 palabras más importantes
X_vectorizado = vectorizador.fit_transform(X)
```

Por último, la otra característica es que para poder usar X_train y X_test hay que transformarlos a formato denso. Esto quiere decir que Keras no trabaja directamente sobre las matrices dispersas que ofrece CountVectorizer por lo que hay que transformarlas a matrices densas de esta manera:

```
X_train = X_train.toarray()     # lo transformamos a formato denso
X_test = X_test.toarray()
```

Para ver cómo funciona internamente esto propongo un pequeño ejemplo. Para las siguiente frases:

```
vectorizador = CountVectorizer()
X_dispersa = vectorizador.fit_transform(["El gato corre", "El perro corre"])
```

Se genera el siguiente vocabulario:

‘corre’	‘el’	‘gato’	‘perro’

Se vectorizan de la siguiente manera en una matriz dispersa de la siguiente forma:

```
Matriz dispersa:
  (0, 1)        1
  (0, 2)        1
  (0, 0)        1
  (1, 1)        1
  (1, 0)        1
  (1, 3)        1
```

Indicando que en la primera frase no aparece la palabra ‘perro’ por lo que (0,3) = 0 dando lugar a este vector:

1	1	1	0

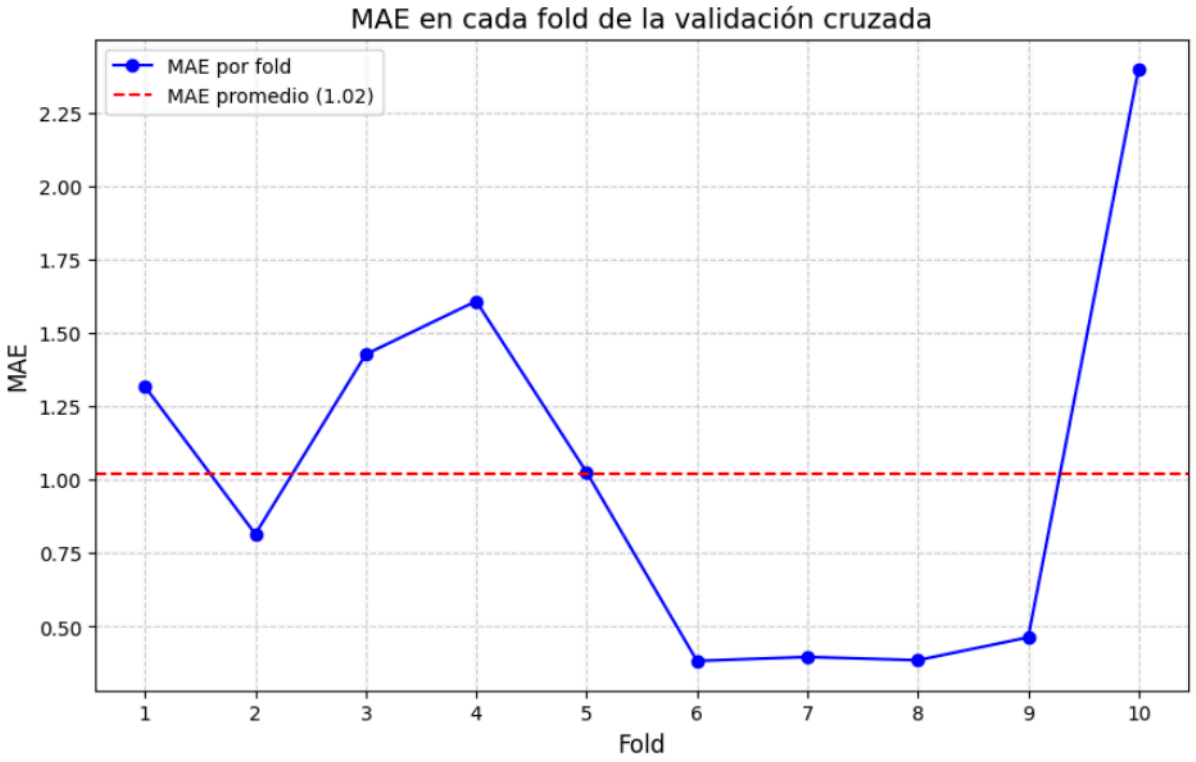
En la segunda frase pasa lo mismo pero con la palabra ‘gato’ por lo que (0,2) = 0 dando lugar a este vector:

1	1	0	1

Dando como lugar esta matriz densa:

```
Matriz densa:
 [[1 1 1 0]
 [1 1 0 1]]
```

Una vez visto todo esto podemos aplicar la validación cruzada a nuestro modelo y nos proporcionará unos valores de MAE. Esta métrica calcula cuánto se desvían las predicciones de los valores reales. Al entrenar nuestro modelo nos da un valor promedio de MAE de 1.0222.. He recogido cada uno de los valores de MAE por Fold de 10-CV y la gráfica resultante es la siguiente:



Esto indica que tenemos un error absoluto medio moderado para un modelo inicial, es decir, nuestro modelo tiene un error promedio de 1 para valores entre 0 y 3 lo cual es mejorable pero aceptable para algo inicial.

Aplicación de un par de redes neuronales.

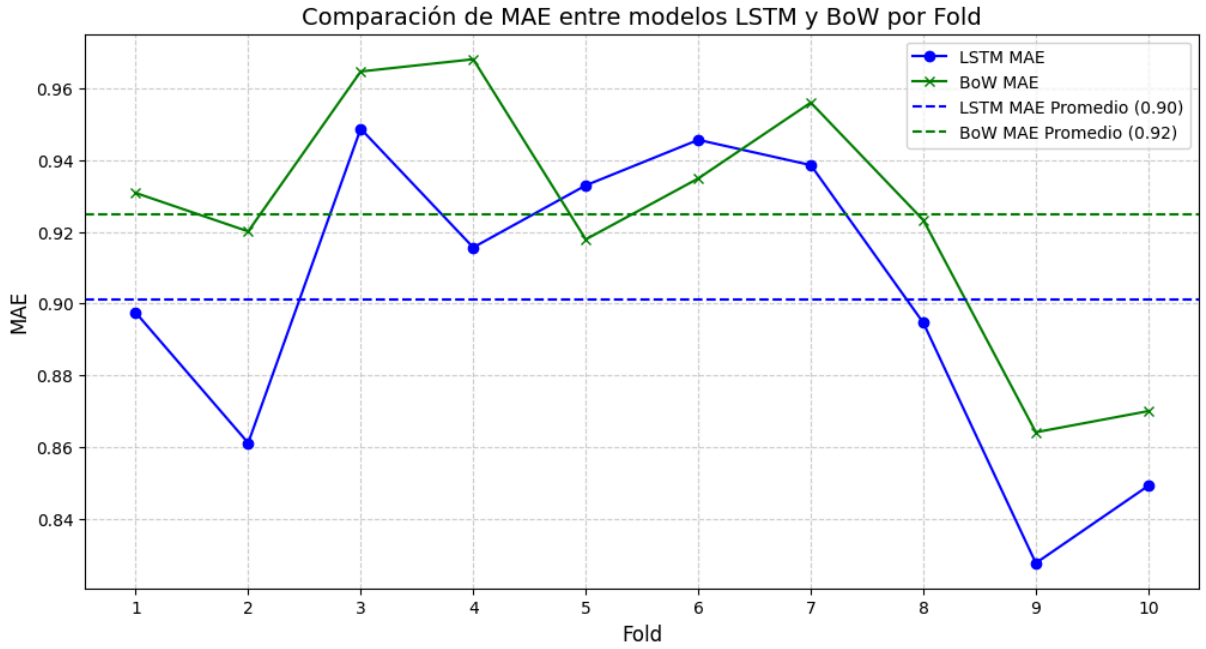
Para realizar esta tarea se han usado dos tipos de redes neuronales, un modelo BoW que es como el que se utilizó anteriormente y un modelo LSTM, estos modelos están basados en secuencias de palabras por lo que preservan el orden y el contexto del texto. Este tipo de modelos son redes neuronales recurrentes (RNN) diseñadas para capturar relaciones contextuales y dependencias entre palabras presentes en el texto. Se caracterizan por:

1) Tokenizador: Las palabras pertenecientes al texto las convierte en índices mediante un diccionario generado por un Tokenizer.
   Por ejemplo: La frase “El pájaro come” se transformaría en un diccionario de esta forma → {“el” : 1, “pajaro” : 2, “come” : 3}
2) Padding: Se rellenan las secuencias para que tengan la misma longitud y se puedan tratar todas por igual.

Por ejemplo, si la longitud máxima de la secuencia es 6 la anterior frase se rellenará de la siguiente forma: → [0, 0, 0, 1, 2, 3]

Este modelo se caracteriza por estar construido por una capa de Embbeding para representar cada una de las palabras como un vector denso, una capa de LSTM, en nuestro caso de 64 unidades, para capturar las dependencias de la palabra a largo plazo, es decir, en las frases “Me senté en el banco” y “Ingresé dinero en el banco” en la primera frase “senté” da el contexto de la palabra y en la segunda “Ingresé” es la que da el contexto a la palabra banco y por último una capa de salida que predice valores continuos con función de activación ‘linear’.

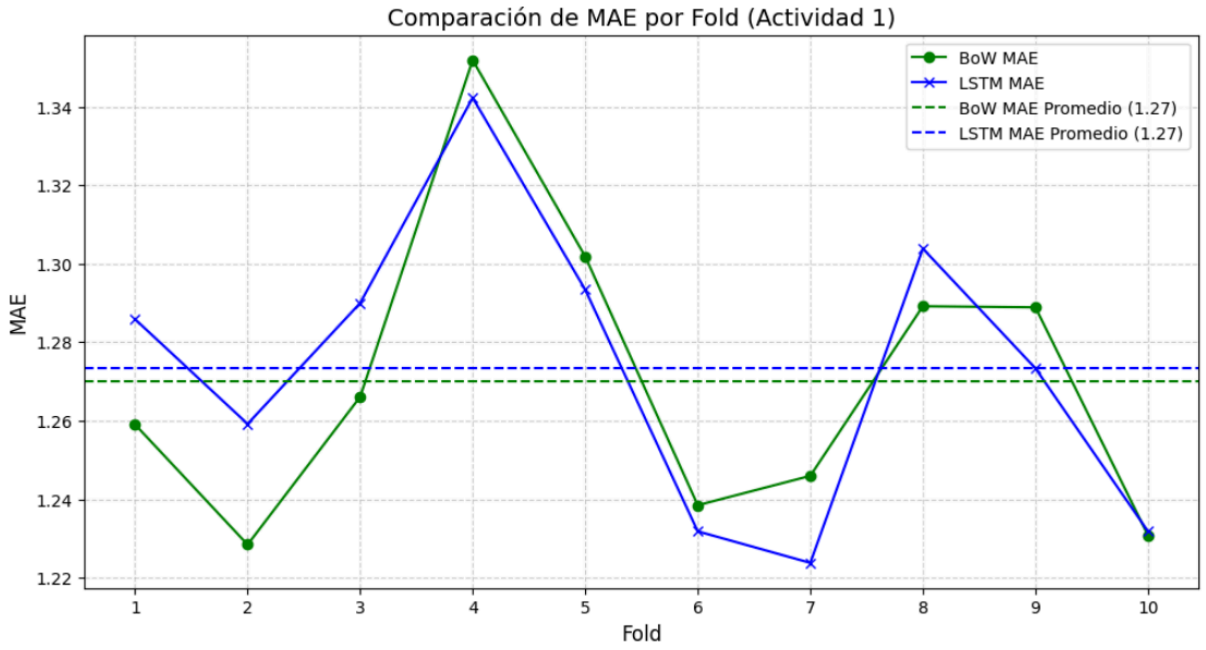
A cada uno de estos modelos se le aplicó validación cruzada (10-CV) y los resultados obtenidos fueron recogidos en la siguiente gráfica:



Como podemos observar en la gráfica el modelo LSTM tiene el valor de MAE ligeramente menor que el de BoW, es decir, en términos generales es un poco más preciso. Además, las líneas entre valores obtenidos se cruzan varias veces por lo que no hay un modelo que sea consistentemente mejor que el otro o al menos no podemos afirmarlo por el momento.

Evaluación de la redes por actividad

Para esta tarea ya que se deja elegir para qué actividad queremos filtrar yo en mi caso he elegido la actividad 1. El proceso de entrenamiento es literalmente igual que antes, aplicamos para cada modelo, LSTM (tokenizar y aplicar secuencias) y BoW (vectorizar) con un solo cambio ya que los datos de entrada son más reducidos ya que antes teníamos todos los datos del conjunto y ahora solo los que pertenecen a la actividad número 1. En ambos modelos se ha aplicado 10-CV (Validación Cruzada) para entrenarlos. Volvemos a realizar el proceso de entrenamiento y obtenemos estos valores de MAE recogidos en esta gráfica:



Como podemos apreciar ambos modelos nos resultan en un MAE promedio de 1.27 frente al promedio anterior que nos daba con todos los datos que era 0.9 para LSTM y 0.92 para BoW, es decir, ambos modelos son muy sensibles a la cantidad de datos que tenemos con los que entrenarlos además de que otra de las razones por las que es más alto de valor de MAE podría ser por la naturaleza de los textos (patrones de lenguaje más complejo).

Test estadístico

En este apartado será necesario hacer uso del test estadístico de Wilcoxon, al solamente tener 2 modelos sobre los que aplicarlo no generaré ninguna gráfica ya que solo nos dará un valor menor de 0.05 en caso de que sí tengan diferencia significativa ambos modelos o un valor mayor o igual de 0.05 en caso contrario. Para aplicarlo cargamos de la biblioteca scipy.stats la función de Wilcoxon signed-rank test para poder evaluar los dos modelos mediante este test. Al aplicarlo nos da el siguiente resultado:

```
No hay diferencia significativa (p = 0.6250)
```

Al recibir este valor de p en el test de Wilcoxon podemos afirmar que no está siendo ningún modelo es significativamente mejor que el otro ya que estadísticamente tienen un rendimiento similar. Esto puede deberse al tamaño reducido del dataset al haber escogido solamente 1 actividad que cuenta con un total de 4735 entradas. Ahora vamos a realizar el test de Wilcoxon con todas las actividades en ambos modelos a ver qué es lo que está pasando. El proceso es el mismo, entrenamos ambos modelos y obtenemos este resultado:

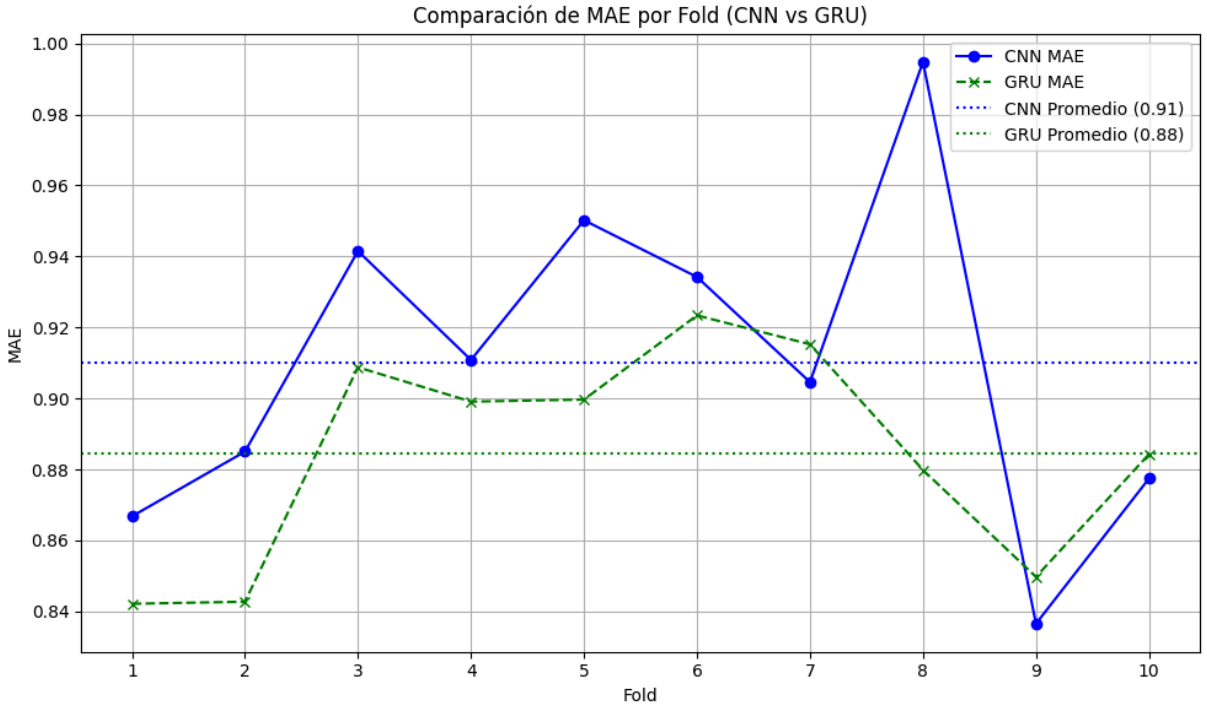
```
Diferencia significativa (p = 0.0098)
```

Esto quiere decir que al obtener un valor de $p < 0.05$ existe una evidencia estadística de que LSTM es mejor que BoW. Aunque aparentemente la mejoría que tiene LSTM sobre BoW es baja como se indicó en dos apartados anteriores aquí se confirma que es claramente mejor que BoW al tener una diferencia estadísticamente significativa.

Medio

Aplicar más arquitecturas de red distintas al apartado básico

El propósito es evaluar otros modelos para ver cómo se desempeñan en la predicción de valores numéricos a partir de texto procesado, se han utilizado dos modelos más, CNN y GRU a parte de los ya presentes que eran BoW y LSTM. Ambas redes fueron entrenadas con 10-CV y se analizaron los valores de MAE en cada entrenamiento. Como datos de entrada ambos modelos se tenían que procesar de la misma manera, se eliminan los caracteres especiales y stopwords (ya procesadas previamente), se aplica un tokenizador para convertir el texto en secuencias de índices y se aplican secuencias que se rellenaron con una longitud de secuencia fija para asegurar que todas las entradas tienen la misma dimensión. El modelo CNN tiene 1 capa de Embedding para representar las palabras como vectores densos, 1 capa convolucional 1D con un kernel para capturar patrones en las secuencias además de 1 capa de MaxPooling1D para reducir la dimensionalidad. Finalmente 1 capa de salida que realiza predicciones continuas. El modelo GRU es similar a la red CNN pero reemplaza la capa de convolución por una capa GRU para capturar dependencias secuenciales en el texto como el orden de las palabras. Al entrenar estos modelos con 10-CV obtenemos una gráfica que representa la diferencia entre estos modelos, la gráfica es la siguiente:



Como podemos observar en la gráfica GRU ofrece una menor variabilidad entre cada fold del 10-CV y además un MAE más bajo, aunque la red CNN tiene un rendimiento peor en este caso puede ser debido al ajuste de los hiperparámetros o con más datos de entrada.

Ajustes combinados

En este apartado vamos a tratar varios aspectos como el número de capas de los modelos definidos anteriormente como red CNN y red GRU, además del tipo de capas de estas redes y normalización. Nuestros modelos inicialmente constaban de esta estructura:

1) Red CNN:

Tipo de capa	Propósito de la capa
Embedding	Representar palabras como vectores densos
Convolucional (kernel)	Extraer características o patrones locales en el texto
MaxPooling1D	Reducir dimensionalidad de las características extraídas
Flatten	Convierte una matriz a unidimensional
Capa densa	Procesa las características extraídas del texto
Dropout	Evita el overfitting
Capa densa de salida	Produce un valor único continuo por regresión

2) Red GRU:

Tipo de capa	Propósito de la capa
Embedding	Representar palabras como vectores densos
GRU	Procesa las secuencias de embeddings para capturar dependencias temporales y contextuales en el texto
Dropout	Evitar overfitting
Capa densa	Aprende relaciones complejas de las características generadas en la capa GRU
Capa densa	Produce la predicción final con un valor continuo

A estos modelos vamos a añadirle más número de capas y con un tipo de capas diferente, es decir, vamos a modificar las redes iniciales de estos modelos para cumplir con el objetivo. Finalmente nuestros modelos quedarán con la siguiente estructura:

1) Red CNN modificada (marcado en gris):

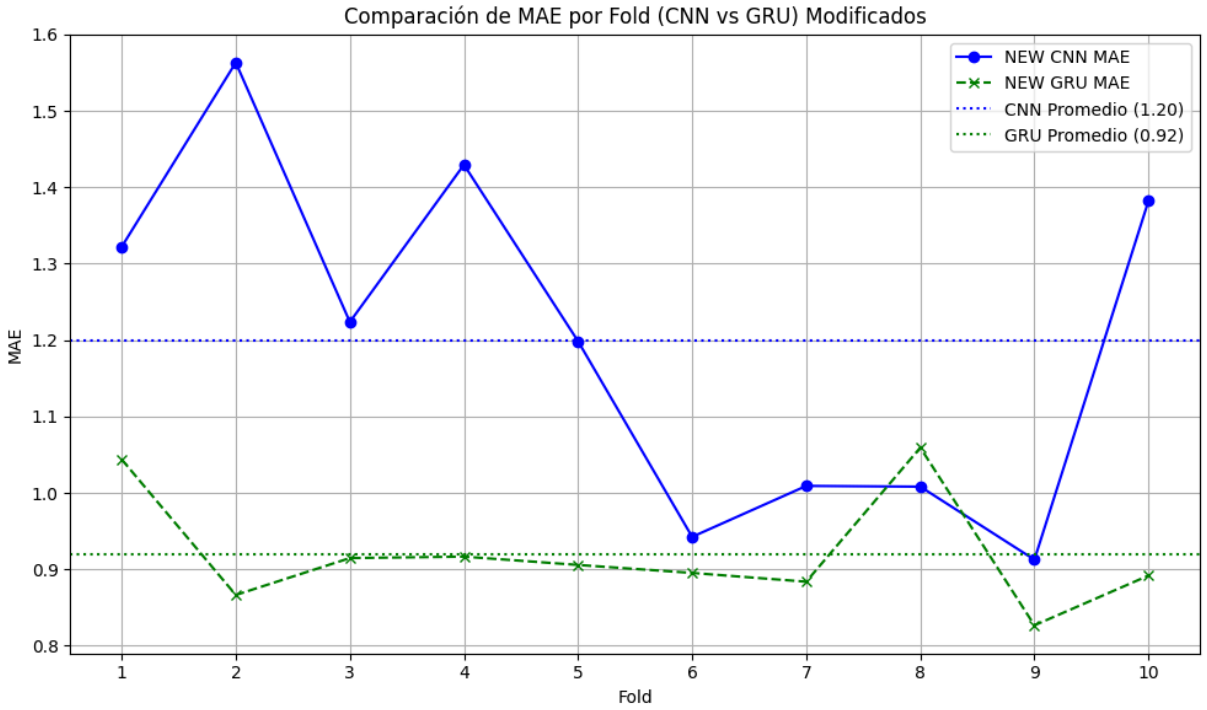
Tipo de capa	Propósito de la capa

Embedding	Representar palabras como vectores densos
Convolucional (kernel)	Extraer características o patrones locales en el texto
BatchNomalization	Aplica normalización para hacer el entrenamiento más estable
MaxPooling1D	Reducir dimensionalidad de las características extraídas
Convolucional (kernel más pequeño)	Segunda capa convolucional (mismo fin)
MaxPooling1D	Reducir dimensionalidad de las características extraídas
Flatten	Convierte una matriz a unidimensional
Capa densa	Procesa las características extraídas del texto
Dropout	Evita el overfitting
Capa densa de salida	Produce un valor único continuo por regresión

2) Red GRU modificada (marcado en gris):

Tipo de capa	Propósito de la capa
Embedding	Representar palabras como vectores densos
Bidirectional	Procesa las secuencias tanto para adelante como para atrás en el tiempo por lo que captura dependencias en ambos sentidos
BatchNomalization	Aplica normalización para hacer el entrenamiento más estable y acelerarlo
Dropout (0.2)	Evitar overfitting y desactiva las neuronas en un 20% de las conexiones durante el entrenamiento
GRU (32)	Procesa las características aprendidas en Bidirectional, produce un vector compacto del texto
Capa densa	Aprende relaciones complejas de las características generadas en las capas GRU
Dropout (0.5)	Evitar overfitting y desactiva las neuronas en un 50% de las conexiones durante el entrenamiento
Capa densa	Produce la predicción final con un valor continuo por regresión

Al tener estos nuevos modelos los he entrenado utilizando 10-CV para ver qué resultados obtenemos sobre estas nuevas modificaciones. Todos estos datos están recogidos en esta gráfica:

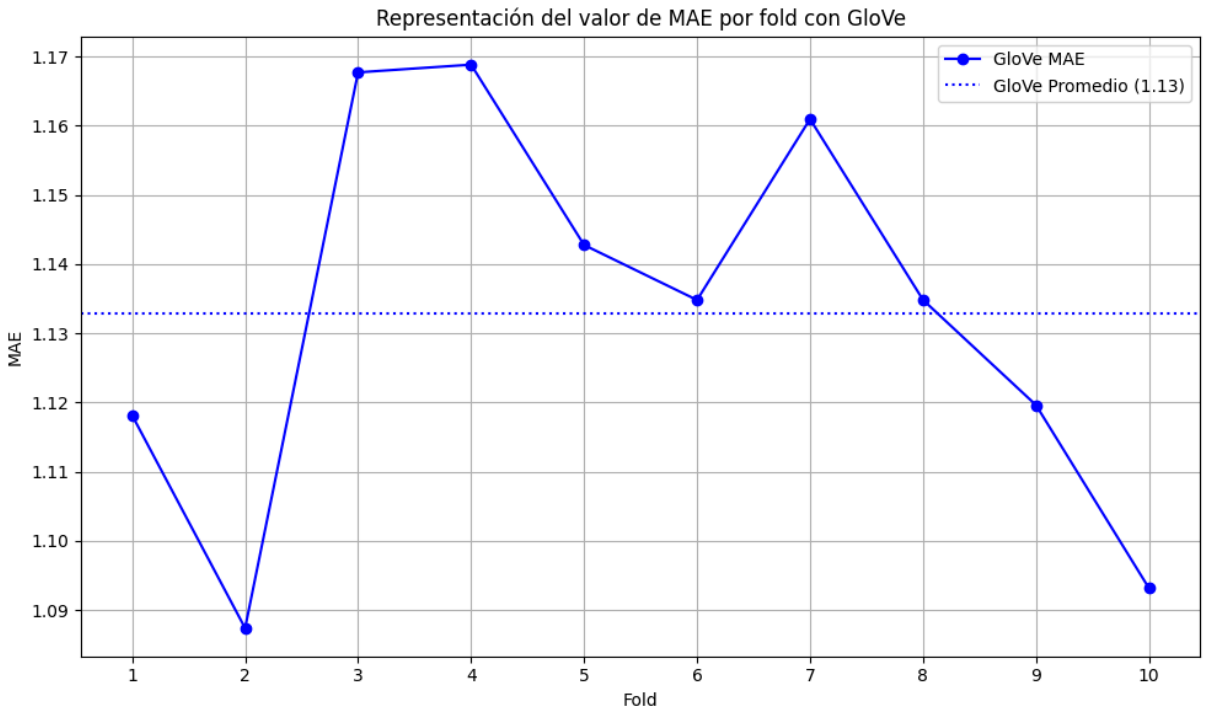


Como podemos observar en la gráfica el rendimiento de la red CNN es significativamente mayor que el de la red GRU, esto quiere decir que su desempeño en el problema con el ajuste de capas e hiperparámetros que se le ha asignado no cumple con el rendimiento que se espera de él por lo que GRU es el modelo que mejor rendimiento nos ofrece siendo éste menos variable entre cada uno de los pliegues del 10-CV.

Uso de redes pre-entrenadas

Para ello voy a hacer uso de la red pre entrenada GloVe. GloVe es un modelo de embeddings de palabras cuyo objetivo es igual al de muchos modelos que ya vimos anteriormente que consta de capturar relaciones semánticas presentes en el texto basándose en estadísticas globales del texto.
Este modelo funciona basándose en las estadísticas globales presentes en el texto como ya comenté, esto significa que palabras que se repiten con frecuencia con misma descendencia semántica tendrán representación similar del espacio vectorial. Para ello hay que construir una matriz de embeddings previa que captura la frecuencia de veces con la que aparece una palabra en el texto y otra ventana con el contexto en el que aparece esa palabra. En cuanto a la descendencia semántica propongo un ejemplo para entenderlo mejor: Las palabras “gato” y “perro” estarán cercanas en el espacio vectorial. En cuanto a las relaciones entre palabras se preservan, es decir, las palabras “rey” + mujer → “reina”.

La gráfica de rendimiento obtenida en el entrenamiento de este modelo es la siguiente:

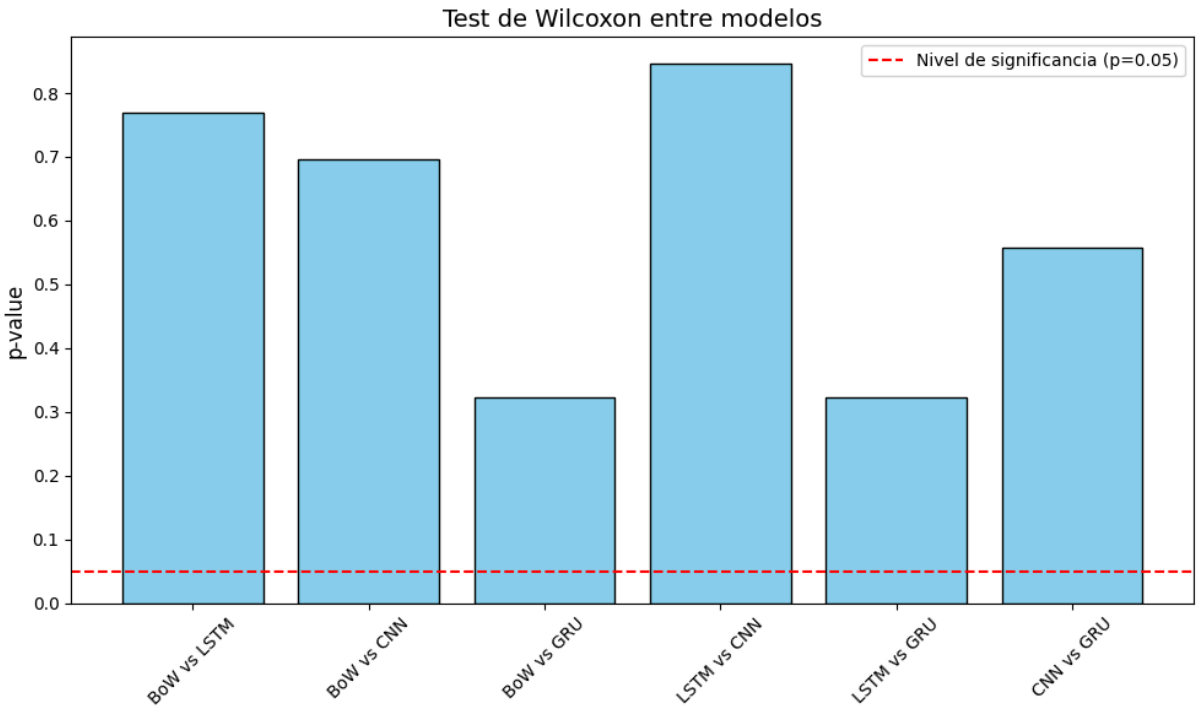


Como podemos observar en la gráfica a comparación del modelo creado anteriormente basado en LSTM o BoW tiene un rendimiento plenamente menor, esto puede ser debido a que este tipo de modelos como GloVe utilizan embeddings estáticos por lo que cada palabra tiene un representación fija independientemente del contexto al que esté asociada por lo que podría estar limitando su capacidad de capturar contextos presentes en cada una de las palabras y no asociarlos correctamente.

Intenté además poner en marcha una red pre entrenada basada en Transformers pero no fui capaz por la falta de entendimiento y por falta de tiempo ya que me daba muchos errores y no logré cómo podría solucionarlo todo :(.

Test estadístico de Wilcoxon

Ya que solamente utilicé una red pre entrenada voy a realizar el test estadístico de Wilcoxon sobre los modelos BoW, LSTM, CNN y GRU para ver cuál de todos da mejores resultados. La gráfica generada analiza por pares cada uno de los modelos y da una representación visual de cuál de todos es significativamente mejor que el resto. La gráfica es la siguiente:



Como podemos observar no hay una evidencia significativa estadística entre cada par de modelos, es decir, todos tienen un rendimiento en general muy similar aunque hay una que destacar entre todas las barras ya que el modelo GRU es el que obtiene en ambos casos el menor nivel de significancia por lo que podríamos decir que por eso y por tener el menor valor promedio de MAE puede ser el mejor de los modelos planteados.

Bibliografía

· Curva ROC:https://www.flickr.com/photos/ligdieli/48049012142/
· ChatGPT para gráficas y generación de ejemplos: https://chatgpt.com (Uso de GloVe)
· Gemini (integrado en Colab) para corrección de errores: https://gemini.google.com/app?hl=es-ES
· StratifiedKFold: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
· StratifiedKFold ejemplo: https://medium.com/@literallywords/stratified-k-fold-with-keras-e57c487b1416
· Test de Wilcoxon (Colab): https://drive.google.com/file/d/1L1_BtjMsd6ReTxn6V0aL9J8si4XR-Evx/view
· ImageDataGenerator doc: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator
· Uso de ResNet50: https://keras.io/api/applications/resnet/
· Bag of Words: https://en.wikipedia.org/wiki/Bag-of-words_model
· MAE: https://es.wikipedia.org/wiki/Error_absoluto_medio
· CNN aplicadas a texto: https://gonzalezmas.es/post/2021-11-23-clasificacion-cnn/
· LSTM: https://medium.com/@rebeen.jaff/what-is-lstm-introduction-to-long-short-term-memory-66bd3855b9ce