Práctica Deep Learning

Predicción del Éxito de Atracciones Turísticas
Un Enfoque de Deep Learning

Sofía Gabián Domínguez

1.- Objetivo

El presente proyecto tiene como objetivo principal desarrollar e implementar un modelo avanzado de Deep Learning que permita predecir con precisión el nivel de engagement que generarán distintos puntos de interés (POIs) turísticos.

Este modelo integrará de manera innovadora dos fuentes de información complementarias: características visuales extraídas de imágenes y metadatos estructurados asociados a cada POI. Los datos utilizados en este estudio provienen de la plataforma Artgonuts, garantizando así su relevancia y aplicabilidad en contextos reales del sector turístico. Las imágenes empleadas han sido específicamente procesadas para los fines de esta práctica, siendo sus versiones originales en alta resolución procedentes de diversas fuentes, incluyendo el portal de datos abiertos de la Comunidad de Madrid.

El desafío central consiste en desarrollar un sistema capaz de anticipar con exactitud el nivel de interacción que cada POI generará, basándose tanto en sus atributos visuales como en sus metadatos descriptivos.

El modelo desarrollado actuará como un clasificador, determinando si un punto de interés turístico (POI) generará un nivel de engagement alto o bajo entre los usuarios. Esta predicción se fundamentará en dos tipos de información complementaria:

- Características visuales: Extraídas mediante redes neuronales convolucionales (CNN) que analizarán patrones, colores, composición y elementos distintivos presentes en las imágenes de cada POI. Estas redes son capaces de identificar automáticamente atributos visuales que resultan atractivos para los usuarios.

- Metadatos estructurados: Información contextual como ubicación geográfica (coordenadas, distrito, barrio), categorización (tipo de atracción, temática), horarios, accesibilidad y otros atributos descriptivos que pueden influir significativamente en el interés que genera cada POI.

Este enfoque híbrido multimodal permite capturar la complejidad del comportamiento del usuario, integrando tanto la respuesta estética inmediata provocada por los elementos visuales como los factores prácticos y contextuales representados en los metadatos. La combinación de ambas fuentes de información en un único modelo proporciona una capacidad predictiva superior a la que se obtendría utilizando cada tipo de datos por separado, constituyendo así una herramienta estratégica para la optimización y personalización del contenido en plataformas turísticas.

2.- Metodología

2.1 Descripción del Dataset

El dataset utilizado contiene información detallada sobre puntos de interés (POIs) turísticos, estructurada de la siguiente manera:

- Imágenes: Cada POI incluye una imagen principal representativa, almacenada en carpetas individuales identificadas por el ID único del punto de interés. La ruta a estas imágenes se encuentra en la columna main_image_path.

• Metadatos: Información contextual sobre cada POI, incluyendo:

  – id: Identificador único para cada punto de interés.
  – name: Nombre descriptivo del POI.
  – shortDescription: Breve descripción textual del punto de interés.
  – Categorización: Columna categories que indica el tipo de atracción o temática.
  – tier: Clasificación por nivel de relevancia o popularidad.
  – Ubicación geográfica : Coordenadas (locationLon, locationLat), barrio, distrito.
  – tags: Etiquetas descriptivas asociadas al POI.
  – xps: Experiencia obtenida por el usuario al visitar el POI (para gamificación).
  – main_image_path : Ruta a la imagen principal de cada POI.

• Métricas de engagement: Indicadores cuantitativos de la interacción de los usuarios con cada POI:

  – Visits: Número de visitas registradas.
  – Likes y Dislikes: Cantidad de valoraciones positivas y negativas recibidas.
  – Bookmarks: Número de veces que el POI ha sido añadido a favoritos.

2.2 Preparación y Análisis de Datos

Antes que nada debemos conocer nuestros datos para poder realizar un correcto preprocesamiento de datos.

Esta es una imagen general del csv.

	0	1	2	3	4	5
id	4b36a3ed-3b28-4bc7-b975-1d48b586db03	e32b3603-a94f-49df-8b31-92445a86377c	0123a69b-13ac-4b65-a5d5-71a95560cff5	390d7d9e-e972-451c-b5e4-f494af15e788	023fc1bf-a1cd-4b9f-af78-48792ab1a294	bcd58127-76bd-44e7-84d8-cc25b46c7962
name	Galería Fran Reus	Convento de San Plácido	Instituto Geológico y Minero de España	Margarita Gil Roësset	Museo del Traje. Centro de Investigación del P...	Clara Campoamor
shortDescription	La Galería Fran Reus es un espacio dedicado a ...	El Convento de San Plácido en Madrid, fundado ...	El Instituto Geológico y Minero de España, sit...	Margarita Gil Roësset, escultora y poetisa esp...	El Museo del Traje de Madrid, fundado en 2004,...	Clara Campoamor fue una abogada, escritora y p...
categories	['Escultura', 'Pintura']	['Patrimonio', 'Historia']	['Ciencia', 'Patrimonio']	['Cultura']	['Patrimonio', 'Cultura']	['Historia', 'Cultura']
tier	1	1	2	1	1	1
locationLon	2.642262	-3.704467	-3.699694	-3.691228	-3.727822	-3.690211
locationLat	39.572694	40.423037	40.442045	40.427256	40.439665	40.435082
tags	[]	[]	[]	[]	[]	[]
xps	500	500	250	500	500	500
Visits	10009	10010	10015	10011	10020	10019
Likes	422	7743	3154	8559	915	8491
Dislikes	3582	96	874	79	2896	71
Bookmarks	78	2786	595	2358	143	2313
main_image_path	data_main/4b36a3ed-3b28-4bc7-b975-1d48b586db03...	data_main/e32b3603-a94f-49df-8b31-92445a86377c...	data_main/0123a69b-13ac-4b65-a5d5-71a95560cff5...	data_main/390d7d9e-e972-451c-b5e4-f494af15e788...	data_main/023fc1bf-a1cd-4b9f-af78-48792ab1a294...	data_main/bcd58127-76bd-44e7-84d8-cc25b46c7962...

Hemos observado que nuestro Data Frame:

a) Tiene unas dimensiones (1569, 14), es decir cuenta con 1,569 filas y 14 columnas.
b) De entrada no cuenta con nulos en ninguna columna.
c) Los tipos que tiene son objet, int64 o float64.

La variable 'tier' es una variable categórica dado que cuenta con 4 valores únicos [1, 2, 3, 4],

d) Las variables 'tags' y 'categoricas' hay que analizarlas un poco más para ver si pueden ser variables categóricas. Parecen ser listas, dado que ambas son listados de atributos que cuenta cada registro.
e) Las variables 'name' y 'shorDescription' son textos.

Empecemos por analizar las posibles variables categóricas.

Para la variable 'categories' nos damos cuenta que tienen 224 valores únicos por registro. Son demasiados valores para considerarlos como variables categóricas tal cual. Lo mismo sucede con la variable 'tags' que cuenta con 1,418.

Al agrupar la 'categories' en los valores únicos en general vemos que son 12 categorías, con esta información si podriamos trabajar con ella como variable categórica. No sucede lo mismo con 'tags', debemos pensar como trabajar este campo.

Observando esta dos variables podemos darnos cuentas que realmente no son listas, sino que son strings. Así que vamos a pasarlos a listas para poder trabajar adecuadamente. Al pasarlas a listas y ver cuantos hay por tamaño, nos damos cuenta que contamos con listas vacias dentro de cada variable (107 para 'tags' y 2 para'categories')

Variable	Minimo	Maximo
tier	1	4
xps	0	1000
Visits	10001	10038
Likes	100	26425
Dislikes	52	10999
Bookmarks	50	8157

Como podeos ver en la tabla anterior observamos los valores en 'Visits', 'Likes', 'Dislikes' y 'Bookmarks' son muy elevados. Cuando tengamos nuestra variable objetivo, debemos tener cuidado de que no pase con valores tan altos. Aunque no tiene valores tan altos debemos preprocesar la variable 'xps'

2.3 Preprocesamiento de datos

2.3.1 Codificación de variables categóricas, manejo de nulos y eliminación de variables

- ✓ Para la variable "tags":

Muestra 107 registros vacios, podríamos agregar una nueva columna que sea el total de tags que se tiene por registro. De esta forma tendremos 0 en los vacios y así podemos trabajar con ellos.

No considero que sera efectivo realizar una imputación ni eliminarnos dado que son bastantes registros y podría darnos información el no tener etiquetas. Y tenemos muchísimos tags para realizarles un one-hot encoding.

- ✓ Para la variable "categories":

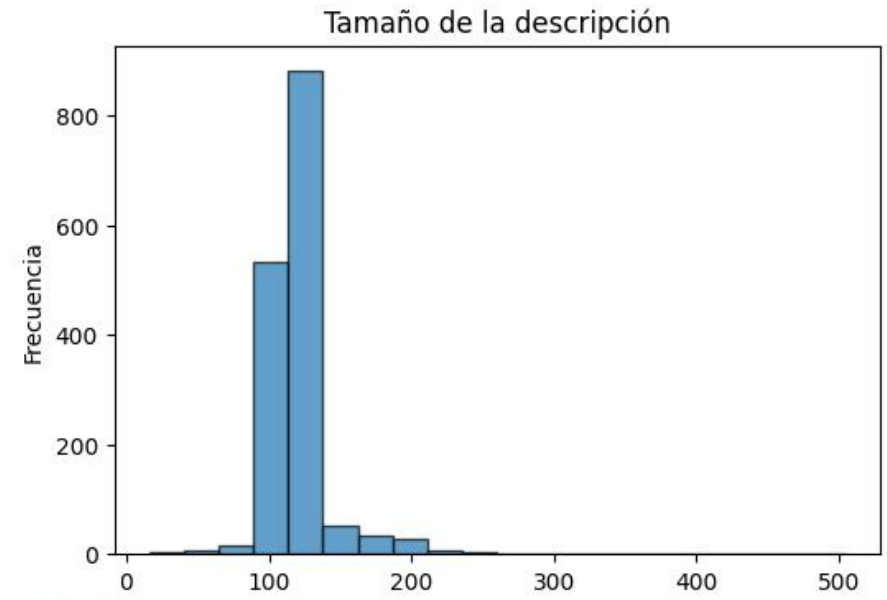
  Tenemos muchísimas convinaciones, por lo que necesitamos realizar One Hot Encoding para asignar valores numéricos a las diferentes categorías de nuestras variables categóricas. Colocando un 1 en la categoría que tenga el registro.

- ✓ Para la variable "tier":

  Aplicaremos un one-hot encoding dado que son pocos varoles únicos, y de esta forma tendremos 1 en el tier al que pertenezca nuestro registro.

Al finalizar podemos eliminar las columnas "tags" y "categories" dado que no trabajaremos con ellas porque son textos y ya contamos con nuevas variables que las representan. De igual manera eliminaremos la variable "tier".

2.3.2 Visualización de nuestros datos

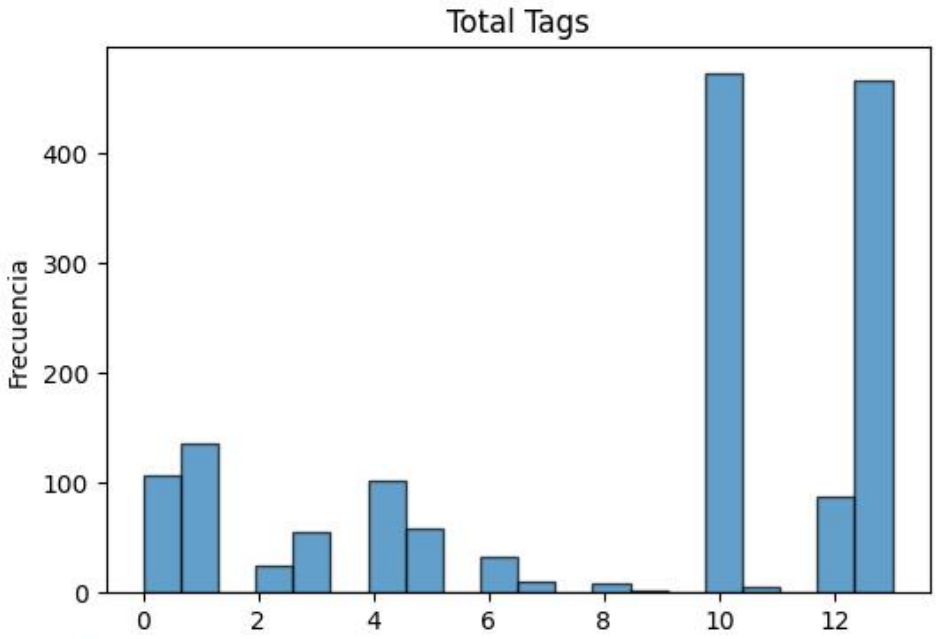


Podemos observar que cuenta con una distribución bastante sesgada a la derecha, lo que nos indica que muchas descripciones son cortas y también van decreciendo como la longitud de las mismas aumenta.

Tenemos un pico muy importante entre 125 y la mayoria se concentran entre 80 y 150.

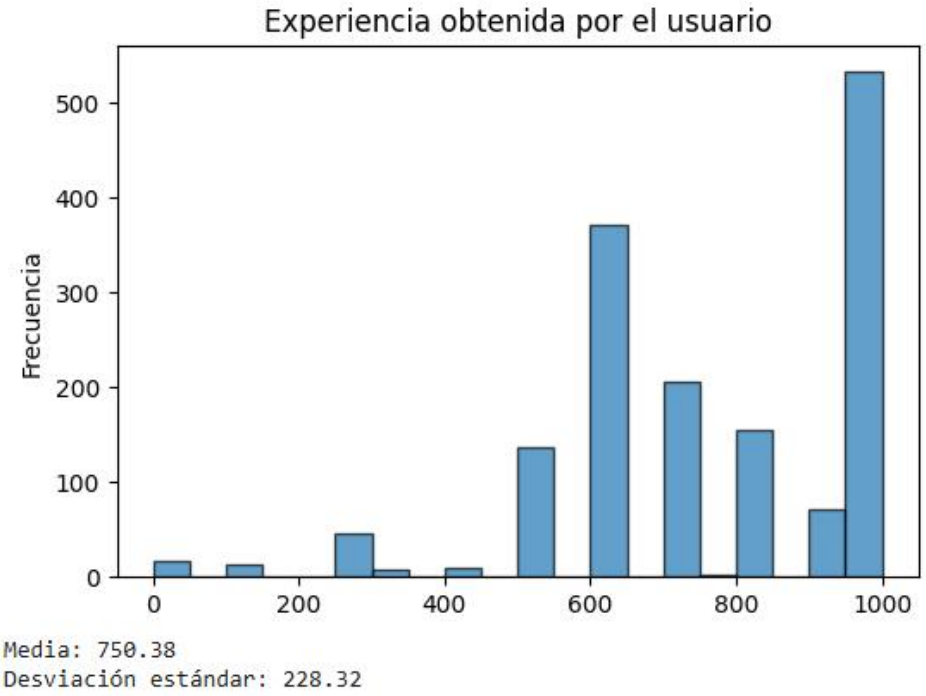
Su distribución parece que nos indica que hay una longitud estándar para las descripción. Por esta razón considero que no es una variable que nos podría aportar información igual que el "id" y el "name".

De media los textos tienen un tamaño de 120 palabras y en el gráfico observamos que son la mayoría.



Parece que esta nueva variable nos puede aportar informacimación dado que muestra una distribución que no es uniforme. Tiene picos significativos en los tags 0, 1,4, 10 y 13 y picos casi nulos en 8, 9 y 11.

De media tienen 8.48 tags pero en la desviación estándar a 4.57 nos damos cuentas que algunos tiene muchos más o muchos menos que la media. Podemos observar que tenemos poco registros en la media.



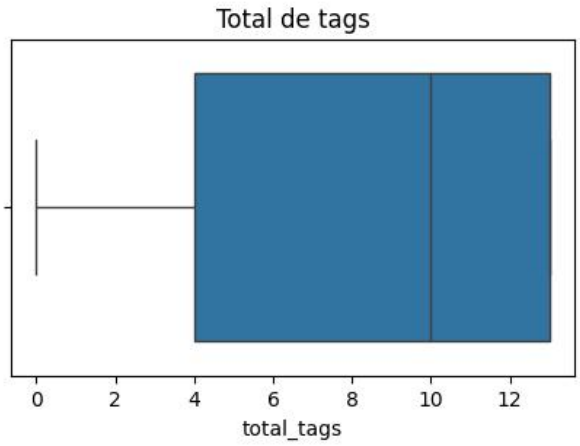
Media: 750.38
Desviación estándar: 228.32

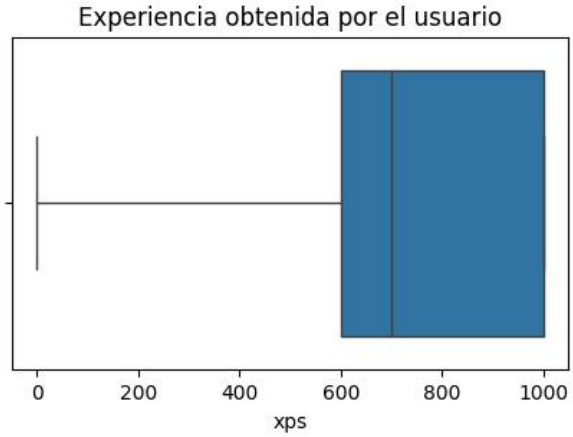
Parece que la mayoria de los puntos tiene una experiencia positiva, podemos observar que la frecuencia disminuye para los valores más pequeños o negativos.

La distribución no es simétrica, vemos que esta sesgada sobre todo hacia valores más alto, como podemos ver en el de 1,000.

Considero que es una variable muy importante que podría mostrar una correlación importante.

Nuestra media es alta 750.38 lo que nos dice que una gran parte de los usuarios tiene niveles de experiencia más elevados. Y como la desviación también es alta observamos como los datos están bastanten dispersos.





En ambos Boxplots podemos observar que no se observan outliers.

- ✓ En "xps" observamos una concentración masiva de usuarios con alta experiencia. Confirmando lo que vimos en el histograma.

- ✓ En "total_tags" podemos ver que la mediana esta alrededor de 10. Los valores del box van desde 4-13, siendo 13 el maximo (como en el histograma).

Ambas variables deben considerarse para el entrenamiento porque sus distribuciones parece que nos puden ayudar a diferenciar los ejemplos.

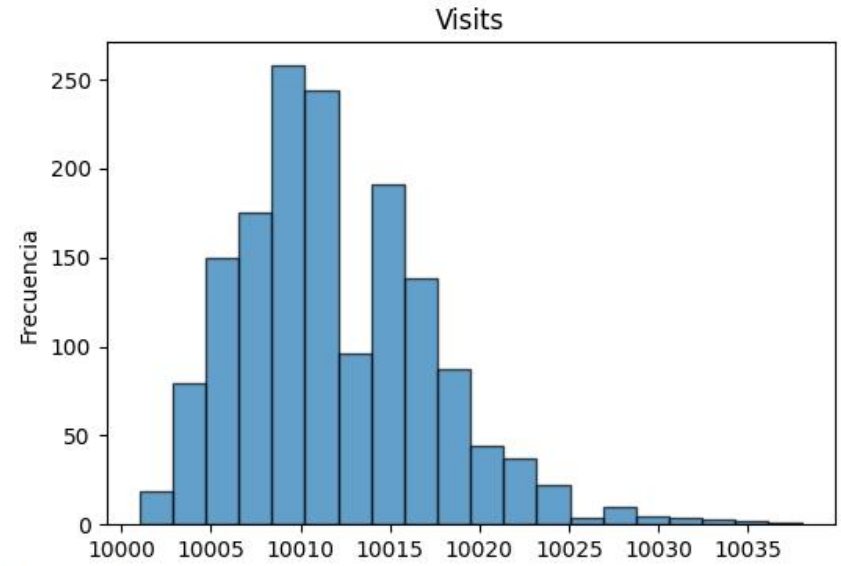
Vamos a elimiar las columnas 'id', 'name', 'len_shortDescription', dado que tienen texto o considero que no aportan información importante a nuestro objetivo.

2.3.3 Creación de métrica de engagement

Esta variable será nuestra variable objetivo, basada en las variables:

- Visits
- Likes
- Dislikes
- Bookmarks

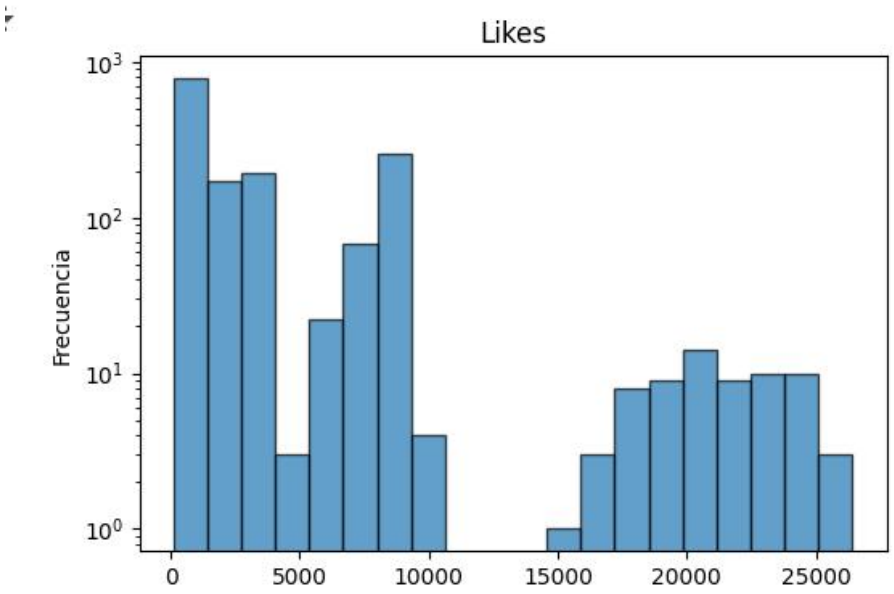
Visualicemos estas variables.



Podemos observar que la mayoría de las visitas están en un rango muy pequeño, aproximadamente entre 10,003 y 10019.

Parece que casi todos los puntos de interés tienen un número de visitas alto.

Podemos observar que la media cae entre los picos mas altos y con la desviación tan pequeña lo comprobamos dado que la mayoria de datos se central alrededor de la media.

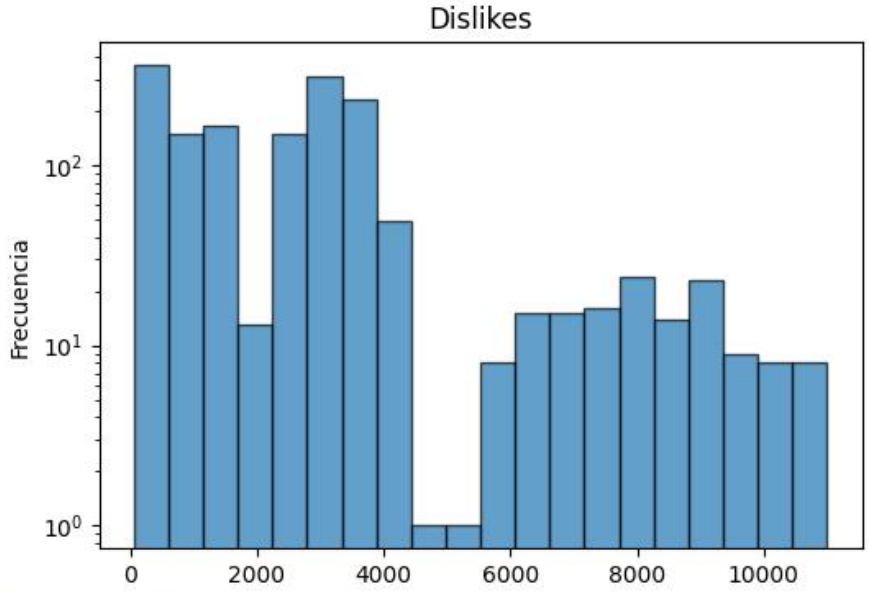



```
Media: 3623.91
Desviación estándar: 4817.88
```

Observamos que la distribución tiene muchos picos, donde parece que existen varios grupos. También vemos sesgo. Para este caso la media no es buena para representarnos el centro de los datos.

Vemos que una gran cantidad de sitios tiene pocos likes (casi 1,000) y casí no se observan frecuencia entre 10,000 y 15,000.

Una desviación estándar tan alta indica muchisima varidad. Los datos están muy dispersos y no se agrupan estrechamente alrededor de la media.

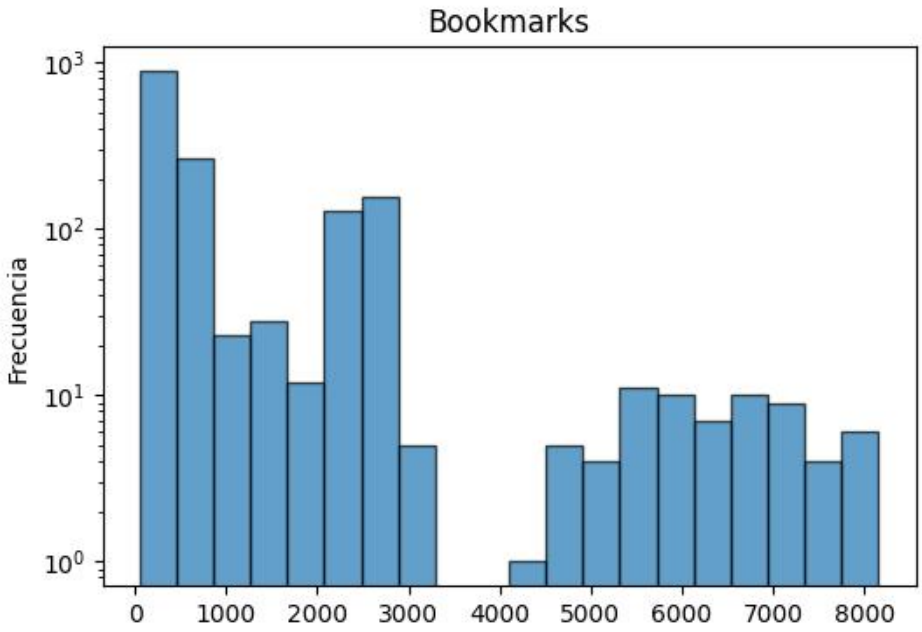



```
Media: 2526.31
Desviación estándar: 2225.54
```

También es una distribución dispersa. Vemos una gran caida alrededor de los 5,000, donde casi no hay registros.

Observamos picos más alto en 0 y aproximadamente 3,000, así como menor frecuencia entre 4,500 y 5,500.

Una desviación estándar tan alta indica muchisima varidad. Los datos están muy dispersos y no se agrupan estrechamente alrededor de la media.




```
Media: 973.26
Desviación estándar: 1453.33
```

Podemos observar un punto muy alto cerca de cero, lo cual nos dice que existen muchos sitios con pocos Bookmarks. Existe un vacio entre los 3,200 y 4,100 aproximadamente que también lo podemos observar en las gráficas anteriores.

Contamos con una distribución irregular y dispersa. En este gráfico también contamos con sesgo importante.

La media es un valor bajo, lo que podemos observa que esta afectada por la gran cantidad e valores cerca del cero. Y nuestra desviación es muy alta en compración con la media lo que nos confirma que hay puntos de interés con muy pocos y otros con muchos Bookmarks.

Vamos a probar con diferentes métricas de engagement.

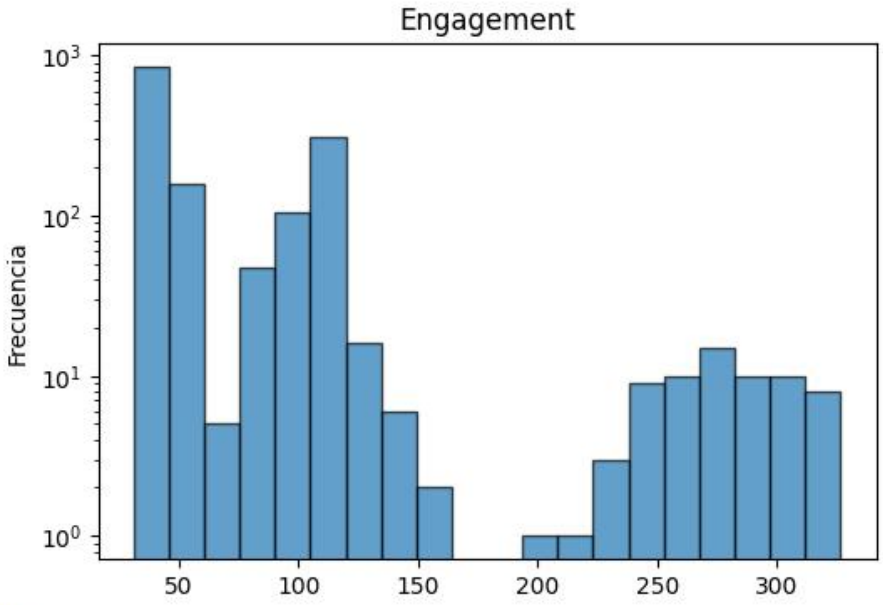
Primera métrica valorando interacciones:

```
engagement = (Número de Interacciones / Alcance) * 100
```

Con nuestra variables quedaría:

```
engagement = ((Likes + Dislikes + Bookmarks) / Visits) * 100;
```

Nota: los Dislikes aunque no son positivos, nos indican una interacción.




```
Media: 71.15
Desviación estándar: 53.71
```

```
Primeras 5 filas con la nueva métrica 1:
   Visits  Likes  Dislikes  Bookmarks  engagement
0   10009    422      3582         78   40.783295
1   10010   7743        96       2786  106.143856
2   10015   3154       874        595   46.160759
3   10011   8559        79       2358  109.839177
4   10020    915      2896        143   39.461078
```

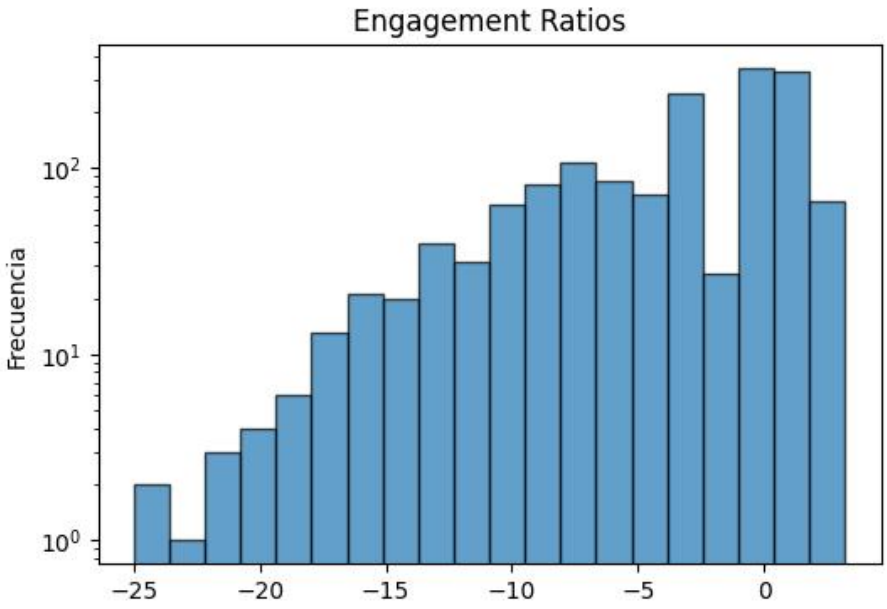
Dado que no tenemos valores negativos podemos observar que al sumar los Dislikes como interacciones no es muy beneficioso. Por ejemplo un punto de interés con muchas visitas (recordemos que los valores son altos) y muchos Dislikes podria tener un engagement alto.

Segunda métrica:

Se busca reflejar qué tan efectivas son las visitas para generar interacciones (Likes, Bookmarks) y cómo se comportan las interacciones negativas (Dislikes) en relación con las positivas.

```
engagement = (Likes / Visits+ϵ) + (Bookmarks / Visits+ϵ) - (Dislikes / Likes+Bookmarks+ϵ)
```

Nota: para evitar divisiones entre cero utilizamos ϵ con un valor muy pequeño. En este caso el valor de epsilon = 1e-8




```
Media: -3.41
Desviación estándar: 5.01
```

```
Primeras 5 filas con las nuevas métricas de ratios y 'engagement_ratios':
   Visits  Likes  Dislikes  Bookmarks  likes_visits  bookmarks_visits  \
0   10009    422      3582         78      0.042162          0.007793
1   10010   7743        96       2786      0.773526          0.278322
2   10015   3154       874        595      0.314928          0.059411
3   10011   8559        79       2358      0.854960          0.235541
4   10020    915      2896        143      0.091317          0.014271

   dislikes_positives  engagement_ratios
0            7.164000          -7.114045
1            0.009118           1.042730
2            0.233129           0.141210
3            0.007236           1.083264
4            2.737240          -2.631651
```

Al calcular los ratios le estamos dando más pesos a la calidad de la interacción en vez de al volumen, por esto vemos muchos valores negativos. Por lo que tendriamos información de mayor calidad.

También podemos observar que se muestra una distribución más gradual (no muestra espacio vacios), esto nos ayudara al momento definir el umbral para engagement alto o bajo.

Tercera métrica:

Otorgaremos diferentes pesos a nuestra variables (según la importancia que les demos) y se aplicara a una formula sencilla donde sumemos las interacciones positivas y restamos la negativa.

```
engagement = (Visits x Wv) + (Likes x Wl) + (Bookmarks x Wb) - (Dislikes x Wd)
```

Definimos los pesos según la importancia que damos:

W_v = debe tener un peso bajo porque como hemos visto en las gráficas anteriores tienen numeros altos.
W_l = normal
W_b = debe ser alto dado que si el usuario decidio guardarlo es que desea verlo en un futuro, por eso tiene más valor que los Likes.
W_d = debe ser más alto porque una opinión negativa siempre tiene más peso que las positivas

Para la primer prueba tenemos los siguientes valores:

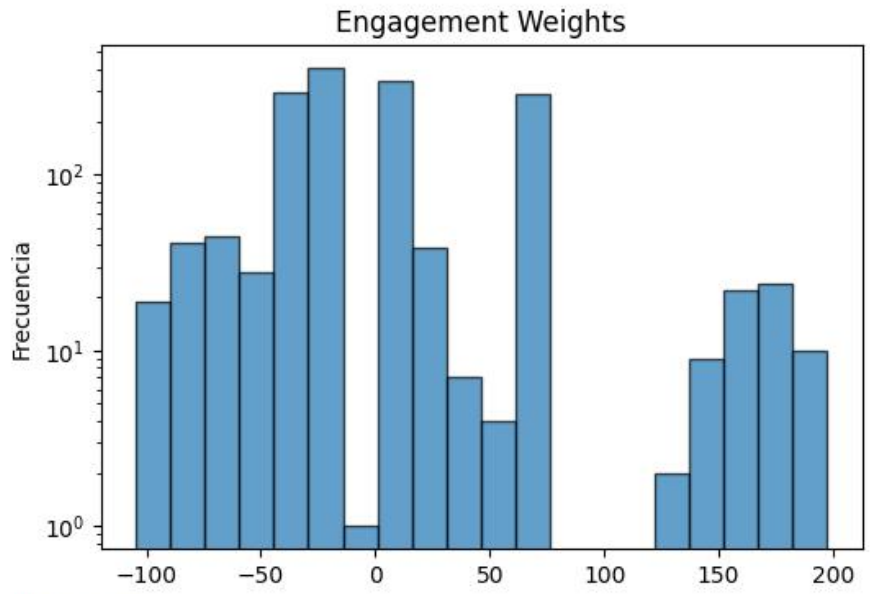
- ✓ W_v = 0.01
- ✓ W_l = 0.5
- ✓ W_b = 1.0
- ✓ W_d = 1.0

Con esta configuración obtuvimos valores muy altos, vamos a reajustarlos.

```
Primeras 5 filas con el nuevo 'engagement_weights':
   Visits  Likes  Dislikes  Bookmarks  engagement_weights
0   10009    422      3582         78            -3192.91
1   10010   7743        96       2786             6661.60
2   10015   3154       874        595             1398.15
3   10011   8559        79       2358             6658.61
4   10020    915      2896        143            -2195.30
--------------------------------------------------
```

- ✓ W_v = 0.0001
- ✓ W_l = 0.005
- ✓ W_b = 0.01
- ✓ W_d = 0.01

```
Primeras 5 filas con el nuevo 'engagement_weights':
   Visits  Likes  Dislikes  Bookmarks  engagement_weights
0   10009    422      3582         78            -31.9291
1   10010   7743        96       2786             66.6160
2   10015   3154       874        595             13.9815
3   10011   8559        79       2358             66.5861
4   10020    915      2896        143            -21.9530
--------------------------------------------------
```



Media: 3.59
Desviación estándar: 53.71

Podemos observar muchos valores negativos, lo que nos dice que para muchos puntos de interés los dislikes o falta de interacciones positivas son la mayoría. Y los que muestran un engagement alto (pocos) nos dice que son muy buenos. Así que las penalizaciones negativas superan las negativas. Y se necesita dominar el área para ajustar los pesos de forma correcta.

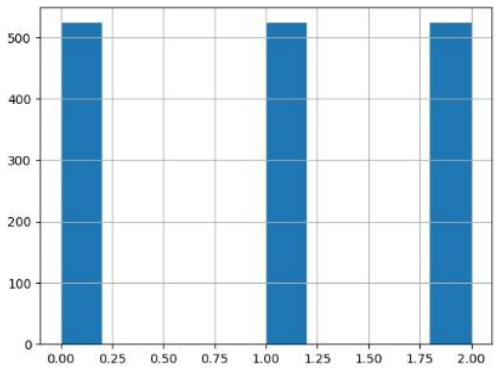
Variable	Minimo	Maximo
Métrica 1	31,00	326,68
Métrica 2	-25,02	3,23
Métrica 3	-104,87	197,50

Nos quedamos con la segunda métrica "**Engagement Ratios**" dado que busca darle más valor a la calidad que al volumen. También penaliza los dislikes en relación a las interacciones positivas en vez de solo restarlos. Y nos tenemos que escalar los resultados.

Vamos a eliminar las variables'Visits', 'Likes', 'Dislikes' y 'Bookmarks dado que nuestra variables objetivo esta creada y no queremos que estas características influyen en nuestro modelo.

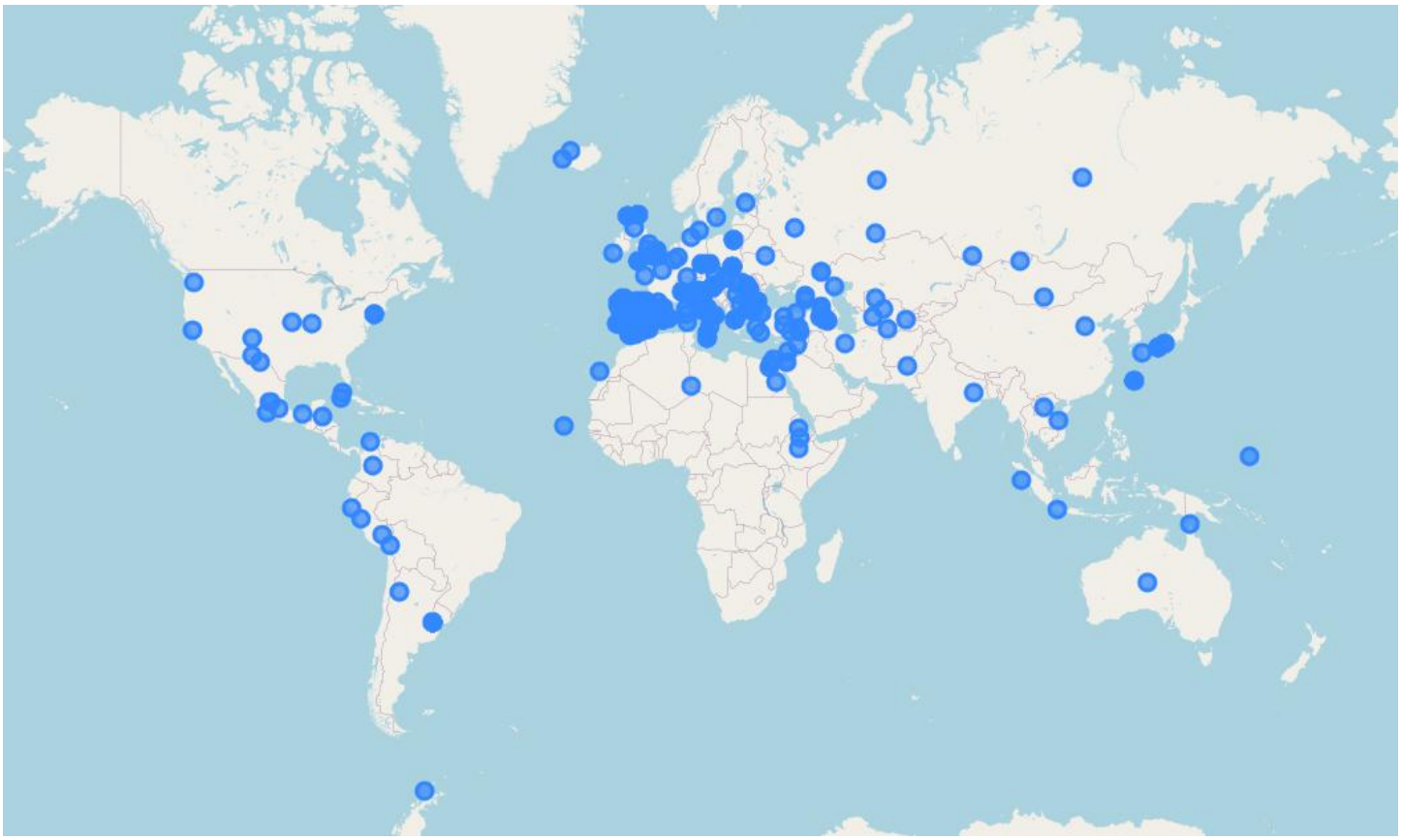
Necesitamos dividir nuestro datos de engagement en grupos que cuenten con aproximadamente la misma cantidad de datos. Los dividiremos en tres grupos: bajo, medio o alto. Quedando iguales.

engagement_scores	count
0	523
1	523
2	523



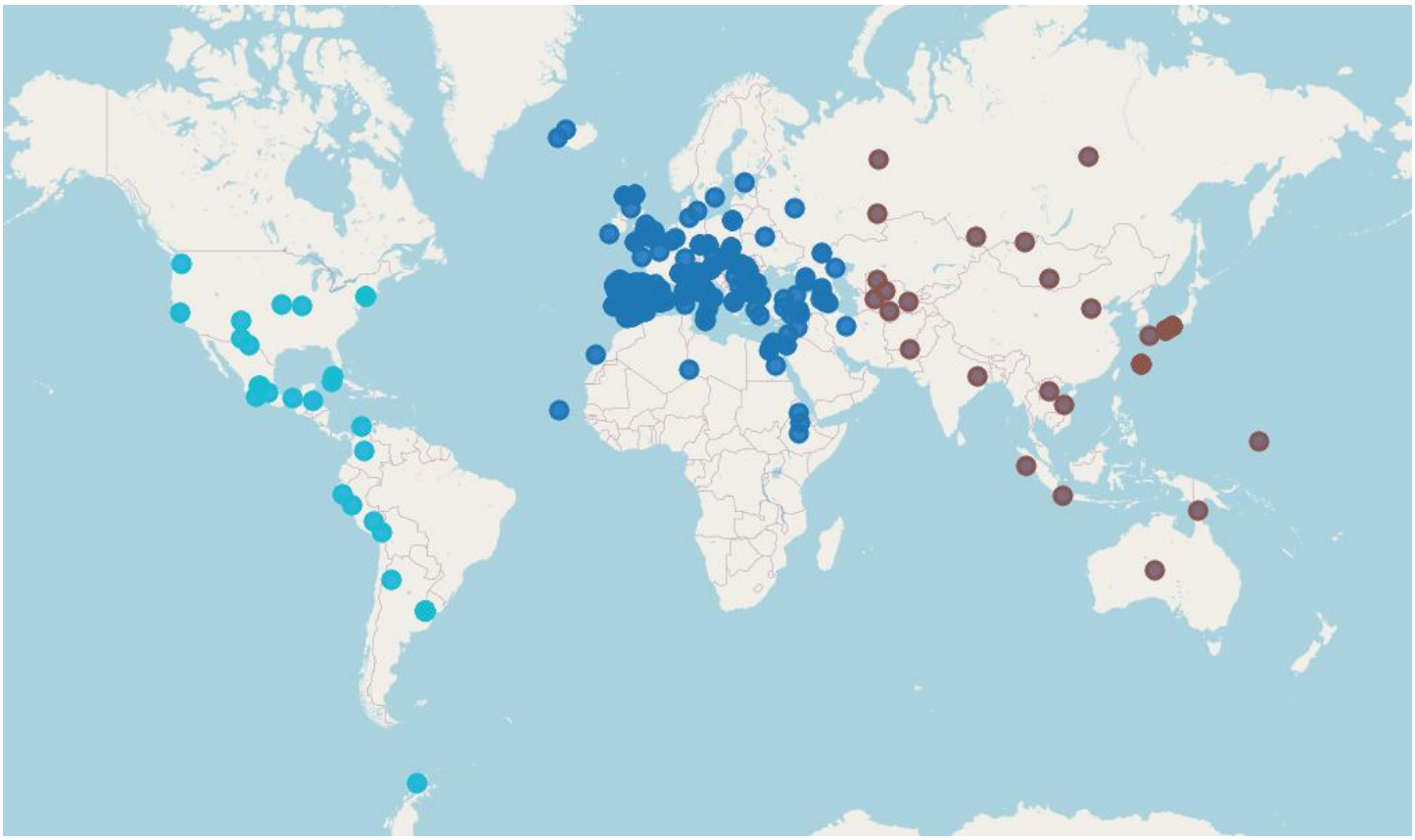
2.3.4 Trabajo con las variables de ubicación

Como cantamos con las variables de 'locationLat' y 'locationLon' podemos saber la ubicación de nuestros puntos de interés.

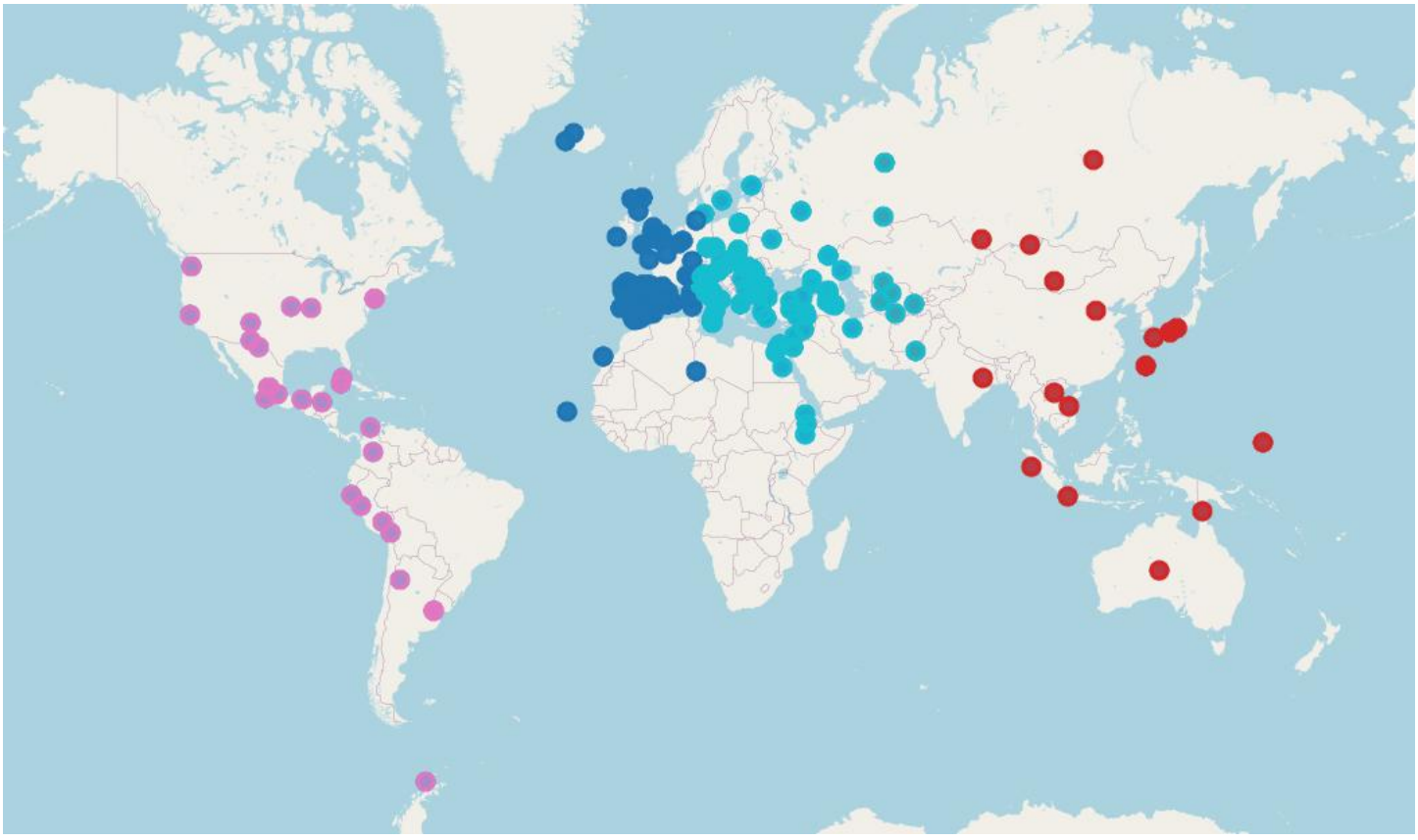


Podriamos agruparlos para utilizar dicha variable en lugar de la latidud y la longitud, así ayudariamos a nuestra red. Y después dicha variable la vamos a tratar como categórica.

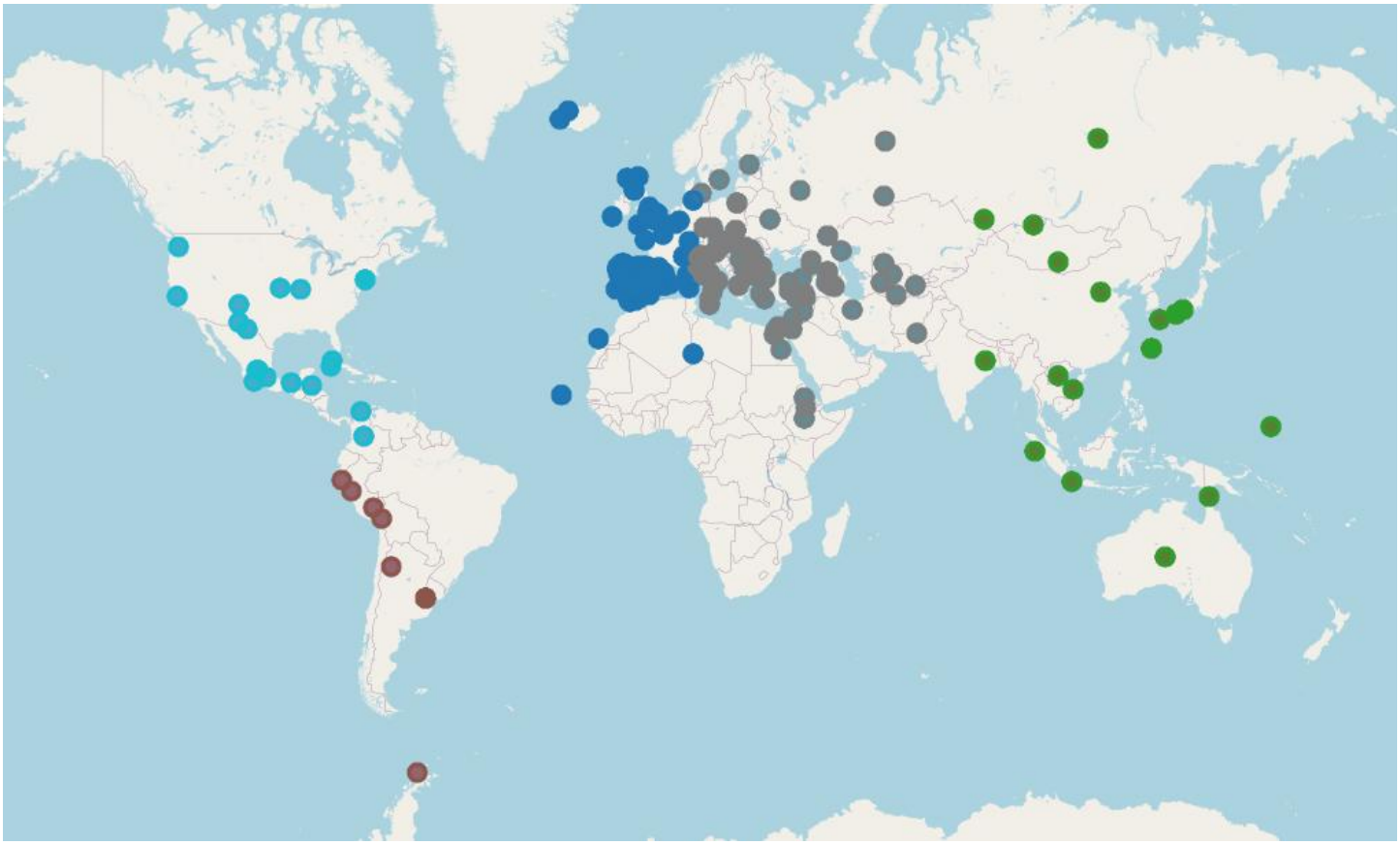
Probaremos primero agrupandolos en 3 clusters.



Ahora agrupandolos en 4 clusters.



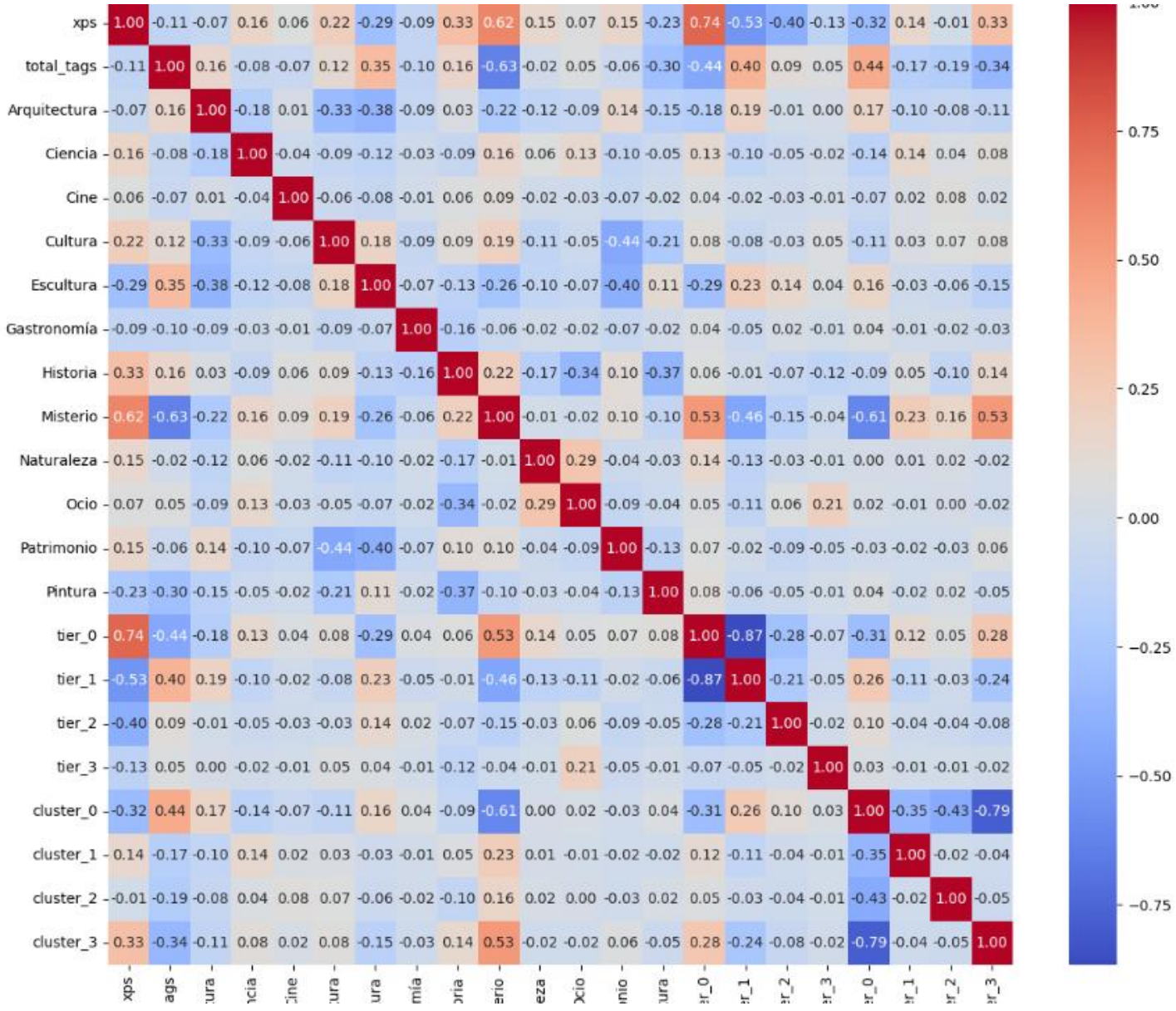
Por último los agruparemos en 5 clusters.



Me voy a quedar con esta cantidad de cluster, dado que con siento que se realiza una buena diferencia entre Europa y Asia con respeto a 3 clusters. Y en comparación con 5 clusters no nos divide América.

Con esta nueva variables lista podremos eliminar las variables 'locationLat' y 'locationLon'. Necesitamos pasarla a una variable categorica aplicando one-hot encoding para que sea más facil para nuestro modelo.

Así quedaría nuestra matriz de correlación

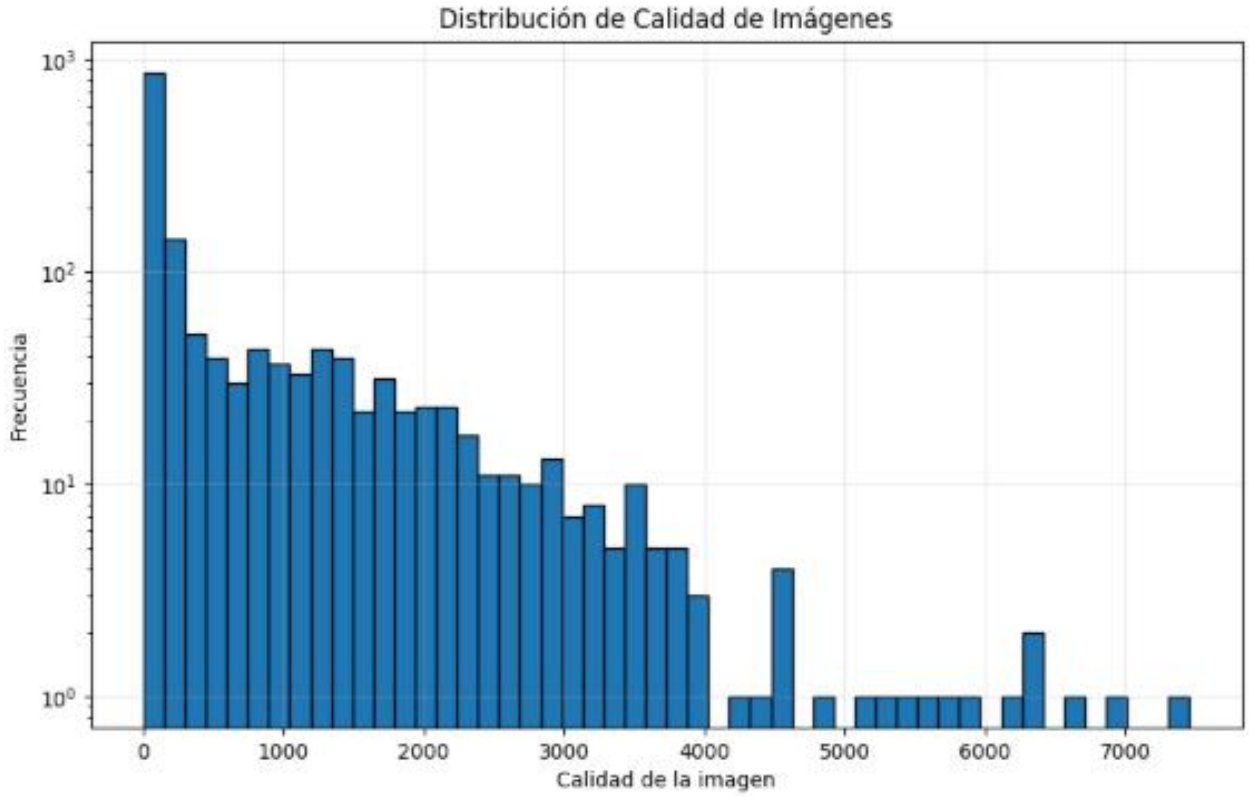


Y así nuestros datos:

	0	1	2
xps	500	500	250
main_image_path	data_main/4b36a3ed-3b28-4bc7-b975-1d48b586db03...	data_main/e32b3603-a94f-49df-8b31-92445a86377c...	data_main/0123a69b-13ac-4b65-a5d5-71a95560cff5...
total_tags	0	0	0
Arquitectura	0	0	0
Ciencia	0	0	1
Cine	0	0	0
Cultura	0	0	0
Escultura	1	0	0
Gastronomía	0	0	0
Historia	0	1	0
Misterio	0	0	0
Naturaleza	0	0	0
Ocio	0	0	0
Patrimonio	0	1	1
Pintura	1	0	0
tier_0	1	1	0
tier_1	0	0	1
tier_2	0	0	0
tier_3	0	0	0
engagement_scores	0	2	2
cluster_0	1	1	1
cluster_1	0	0	0
cluster_2	0	0	0
cluster_3	0	0	0

2.3.5 Imágenes

Todas las imágenes tienen un tamaño(128, 128, 3), es decir, 128 x 128 a color y contamos con imágenes de muy buena calidad así como otras con pésima. Lo que predormina son las de pésima calidad como podemos observar en la gráfica.



Necesitamos preprocesar nuestra imágenes:

- ✓ Primero necesitamos pasar a formato RGB para tener el orden correcto
- ✓ Redimensionales y normalizarlas pasandolas a tensor.
- ✓ Al pasar a tensor convierte la imagen a un tensor de tipo float32 y escala los valores de píxel dividiéndolos por 255, es decir, pasando de enteros 0–255 a flotantes 0.0–1.0.

2.3.6 Dataset personalizado y DataLoaders

Nuestro Dataset debe contener:

- ➢ Las variables correspondientes a nuestras feactures: las diferentes categoria, xps, el número total de tagas, el tier así como el cluster.
- ➢ La path de la imagen: main_image_path
- ➢ Nuestra variable objetivo: engagement_scores

Voy a incluir una parámetro ‘is_training’ para identificar que tipo de transformación vamos a hacer sobre nuestra imágenes, es decir si son para el conjunto de entrenamiento o para el de validación o test.

```python
class POIDataset(Dataset):

  # Función de inicialización
  def __init__(self, features, image_path, target, is_training=False):

    self.feature_cols = features.columns
    self.features = torch.tensor(features.values, dtype=torch.float32)
    self.target = torch.LongTensor(target) #Para que no nos de problemas con el float32
    self.image_dir = image_path
    self.is_training = is_training


  # Función para obtener el tamaño del dataframe
  def __len__(self):
    return len(self.features)


  # Función para obter la imagen
  def __getitem__(self, idx):

    # Obtener el target
    target = self.target[idx].to(device)

    # Obtener las features
    features = self.features[idx].to(device)

    # Cargar y transformar la imagen
    path_img = ruta_imgs + self.image_dir[idx]
    image = cv2.imread(path_img)

    if image is None: # Comprobar si cv2.imread falló
        raise FileNotFoundError(f"No se pudo cargar la imagen: {path_img}. Verifica la integridad del archivo.")
        dummy_image = np.zeros((224, 224, 3), dtype=np.uint8)
        imgen = dummy_image

    # Convertir la imagen de OpenCV (array NumPy) a una imagen PIL
    image = Image.fromarray(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

    # Aplicar la transformación de imagen según si es entrenamiento o no
    if self.is_training:
        image = image_transform_train(image).to(device)
    else:
        image = image_transform_test_val(image).to(device)

    return features, image, target
```

Para nuestro DataSet personalizado vamos a aplicar el pre-procesamiento de ciertos datos:

- ✓ En imágenes nos dividiremos:

  a) Datos de entrenamiento queremos realizar transformaciones con data augmentation, esto por los pocos datos con los que contamos y su mala calidad.

  - ✓ Invertimos la imagen horizontalmente de izquierda a derecha con transforms.RandomHorizontalFlip(p=0.5)
  - ✓ Aplicamos variaciones aleatorias de color a la imagen. Con esto ayudamos al modelo a que sea más robusto a condiciones de iluminación y colores diferentes. Con transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2)
  - ✓ Rotación aleatoria hasta +/- 10 grados. Con transforms.RandomRotation(degrees=10)

- ✓ Aplicar una traslación, es decir mueve la imagen horizontal y verticalmente un 10%. Con transforms.RandomAffine(degrees=0, translate=(0.1, 0.1))
- ✓ Vamos a normalizar con la media y desviación estándar que calculamos del conjunto de datos de entrenamiento

```
Media del conjunto de entrenamiento: 0.43927475810050964
Desviación estándar del conjunto de entrenamiento: 0.26631322503089905
```

```
# Definimos las transformaciones finales para entrenamiento (con data augmentation)
image_transform_train = transforms.Compose([
    transforms.Resize(img_size),
    # Invertimos la imagen horizontalmente de izquierda a derecha.
    transforms.RandomHorizontalFlip(p=0.5),
    # Aplicamos variaciones aleatorias de color a la imagen. Con esto ayudamos al modelo a que sea más robusto a condiciones de iluminación y colores diferentes.
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    # Rotación aleatoria hasta +/- 10 grados
    transforms.RandomRotation(degrees=10), # Rotación aleatoria hasta +/- 10 grados
    # Aplica una traslación, es decir mueve la imagen horizontal y verticalmente un 10%
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    #Con la media y desviación estandar que calculamos del conjunto de datos de entrenamiento
    transforms.Normalize(mean=mean_train, std=std_train)
])
```

b) Datos de validación y de test, unicamente normalizaremos con on la media y desviación estándar que calculamos del conjunto de datos de entrenamiento.

```
# Definimos transformaciones finales para validación y test a nuestra imágenes
image_transform_test_val = transforms.Compose([
    transforms.Resize(img_size),
    transforms.ToTensor(),
    #Con la media y desviación estandar que calculamos del conjunto de datos de entrenamiento
    transforms.Normalize(mean=mean_train, std=std_train)
])
```

c) Para nuestra features, solo se aplicara a las variables 'xps' y 'total_tags' , dado que todas las demas tiene valores entre 0-1. En el caso del grupo de entrenamiento se aplicara el entrenamiento y la transformación pero para test y validación solo la transformación.

- ✓ Escogi StandardScaler dado que transforma los datos para que tengan una media de 0 y una desviación estándar de 1.

- ✓ Y en el caso de la variable ‘xps’ que cuenta con valores tan altos aplicaremos una base logaritmica.

```python
#Necesitamos escalar algunas variables de nuestros metadatos.

# Lista de columnas a incluir, son solo las que necesitan ser escalonadas.
incluir = ['xps', 'total_tags']
numeric_cols = [col for col in df.columns if col in incluir]

# Procesar datos de entrenamiento
# Concatenar las características escaladas de vuelta al DataFrame de entrenamiento
train_features_scaled = process_features(train_df[numeric_cols], is_training=True)
train_df = pd.concat([train_df.drop(columns=numeric_cols), train_features_scaled], axis=1)

# Procesar datos de validación
val_features_scaled = process_features(val_df[numeric_cols], is_training=False)
val_df = pd.concat([val_df.drop(columns=numeric_cols), val_features_scaled], axis=1)

# Procesar datos de test
test_features_scaled = process_features(test_df[numeric_cols], is_training=False)
test_df = pd.concat([test_df.drop(columns=numeric_cols), test_features_scaled], axis=1)
```

Ejemplo del resultado para el grupo de entrenamiento

xps	total_tags
1.092671	-0.787728
-0.241377	0.317393
-0.686059	0.759442
1.092671	-1.008753
0.203306	0.980466

```
Rango de xps: -3.354153871473131 - 1.092670767397753

Rango de total_tags: -1.8928502497762756 - 0.9804663872385408
```

Los datos fueron divididos de las siguiente forma:

```
Tamaño inicial del DF: 1569
Tamaño del DF de entrenamiento: 1133
Tamaño del DF de validación: 200
Tamaño del DF de test: 236
```

3.- Arquitectura del Modelo

Vamos a experimentar con diferentes arquitecturas, cada una debe ser modelo híbrido que combine:

- ✓ Componente visual: Red convolucional para procesamiento de imágenes (propia o adaptada)
- ✓ Componente contextual: Procesamiento de metadatos mediante capas fully-connected
- ✓ Mecanismo de fusión de ambas ramas de información

3.1 Primer modelo

Este modelo incluirá:

- ✓ Red para features full connected (2 capas con salidas 64 y 32)
- ✓ Red para imágenes trabajaremos con ResNet-18
- ✓ Para la red que las combina full connected (2 capas con salidas 128 y 64)
- ✓ Se inicializaran los pesos con kaiming_normal_, porque va muy bien con la función de activación Relu. Logra tener un entrenamiento más rapido dado que previene los problemas del gradiente que desaparece.

```
# Para nuestra features trabajaremos con capas full connected
self.features_fc = nn.Sequential(

  # Primera Capa full connected
  # BatchNorm1d nos ayuda normalizar la salida de la capa antrior y acelerar el entrenamiento.
  # Función de activación Relu dado que para en general funicona muy bien para clasificación.
  # Ayuda con el problema del gradiente que tiende a desaparecer
  # Dropout para reducir el sobreajuste (overfitting). Nos ayuda a generalizar.
  nn.Linear(input_size, 64),
  nn.BatchNorm1d(64),
  nn.ReLU(),
  nn.Dropout(dropout_rate),

  # Segunda Capa full connected
  nn.Linear(64, 32),
  nn.BatchNorm1d(32),
  nn.ReLU(),
  nn.Dropout(dropout_rate)

)
```

```
# Para nuestra imágenes como no tiene buena calidad ni tenemos muchos datos,
# vamos a provechar la arquitectura de ResNet-18
# Cagamos el modelo pre-entrenado
resnet18 = torchvision.models.resnet18(weights=models.ResNet18_Weights.DEFAULT)

# Necesitamos congelar capas (Transfer Learning)
# freeze_layers = cuántas capas iniciales se van a congelar para así tengamos
# una red mas robusta dado que protege las lo aprendido en las capas congeladas.
for param in list(resnet18.parameters())[:freeze_layers]:
    param.requires_grad = False # No queremos que actualice los pesos de estas capas (congelarlas)

# Para nuestra imagenes trabajaremos con CNN
self.images_cnn = nn.Sequential(
    # Tenemos que eliminar la última capa, que es el clasificador para ResNet que nosotros no vamos a usar
    *list(resnet18.children())[:-1],
    # Aplanamos la salida para poder combinarla.
    nn.Flatten()
)
```

```
        # Obtener el tamaño de salida de la CNN después de Flatten.
        with torch.no_grad(): # No calcular gradientes porque solo vamos a comprabar el la salida de CNN
            ejem_image_input = torch.randn(1, 3, 224, 224)
            cnn_output_dim = self.images_cnn(ejem_image_input).shape[1]
            print(f"Dimensión de salida de images_cnn: {cnn_output_dim}")

        # Red final combinando la de features y la de imagenes y con el clasificador final
        # La salida de features_fc = 32.
        # La salida de images_cnn = 512 deberia de ser porque es lo que saca ResNet18.
        final_input_size = 32 + cnn_output_dim

        self.engagement_classifier = nn.Sequential(
            nn.Linear(final_input_size, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(dropout_rate),

            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.Dropout(dropout_rate),

            nn.Linear(64, num_classes) # Capa de salida para 3 clases
        )

        #Inicialización de pesos y bias
        self.apply(self._initialize_weights_kaiming)
```

```
#Función para inicializar los peos y bias en nuestras capas full connected
def _initialize_weights_kaiming(self, m):
    if isinstance(m, nn.Linear):
      # Utilizamos a Kaiming normal porque va muy bien con la función de activación Relu.
      # Logra tener un entrenamiento más rapido dado que previene los probelmas del gradiente que desaparece.
        nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
        # Y si tiene bias los inicialemos con cero para que no se introduzca un sesgo en las activaciones.
        if m.bias is not None:
            nn.init.constant_(m.bias, 0)

def forward(self, features, images):

    # Pasar las features por la red fc
    features_output = self.features_fc(features)

    # Pasar las imágenes por la CNN
    images_output = self.images_cnn(images)

    # Concatenar las salidas de ambas redes
    # Dim=1 para poder mantener la dimensión del batch.
    output = torch.cat((features_output, images_output), dim=1)

    # Pasar la salida combinada por el clasificador final
    final_output = self.engagement_classifier(output)

    return final_output
```

3.2 Segundo modelo

Vamos a crear una arquitectura más profunda.

Este modelo incluira:

- ✓ Red para features full connected (3 capas con salidas 128, 64 y 32)
- ✓ Red para imágenes trabajaremos con ResNet-50
- ✓ Para la red que las combina full connected (3 capas con salidas 128, 64 y 32)

Nota: añado únicamente los cambios realizados

```
# Para nuestra features trabajaremos con capas full connected
self.features_fc = nn.Sequential(

  # Primera Capa full connected
  # BatchNorm1d nos ayuda normalizar la salida de la capa antrior y acelerar el entrenamiento.
  # Función de activación Relu dado que para en general funicona muy bien para clasificación.
  # Ayuda con el problema del gradiente que tiende a desaparecer
  # Dropout para reducir el sobreajuste (overfitting). Nos ayuda a generalizar.
  nn.Linear(input_size, 128),
  nn.BatchNorm1d(128),
  nn.ReLU(),
  nn.Dropout(dropout_rate),

  # Segunda Capa full connected
  nn.Linear(128, 64),
  nn.BatchNorm1d(64),
  nn.ReLU(),
  nn.Dropout(dropout_rate),

  # Tercer Capa full connected
  nn.Linear(64, 32),
  nn.BatchNorm1d(32),
  nn.ReLU(),
  nn.Dropout(dropout_rate),
)
```

```
# Para nuestra imágenes como no tiene buena calidad ni tenemos muchos datos,
# vamos a provechar ahora con la arquitectura de ResNet-50 que es más profunda
# Cagamos el modelo pre-entrenado
resnet50 = torchvision.models.resnet50(weights=models.ResNet50_Weights.DEFAULT)
```

```
# Red final combinando la de features y la de imagenes y con el clasificador final
# La salida de features_fc = 32.
# La salida de images_cnn = 2048 deberia de ser porque es lo que saca ResNet50.
final_input_size = 32 + cnn_output_dim

self.engagement_classifier = nn.Sequential(
    nn.Linear(final_input_size, 128),
    nn.BatchNorm1d(128),
    nn.ReLU(),
    nn.Dropout(dropout_rate),

    nn.Linear(128, 64),
    nn.BatchNorm1d(64),
    nn.ReLU(),
    nn.Dropout(dropout_rate),

    nn.Linear(64, 32),
    nn.BatchNorm1d(32),
    nn.ReLU(),
    nn.Dropout(dropout_rate),

    nn.Linear(32, num_classes) # Capa de salida para 3 clases
)
```

3.2 Tercer modelo

Vamos a crear una arquitectura una mezcla de ambas.

Este modelo incluira:

- ✓ Red para features full connected (1 capa con salida 32)
- ✓ Red para imágenes trabajaremos con ResNet-50
- ✓ Para la red que las combina full connected (2 capas con salidas 64 y 32)

Nota: añado únicamente los cambios realizados

```
# Para nuestra features trabajaremos con capas full connected
self.features_fc = nn.Sequential(

  # Primera Capa full connected
  # BatchNorm1d nos ayuda normalizar la salida de la capa antrior y acelerar el entrenamiento.
  # Función de activación Relu dado que para en general funicona muy bien para clasificación.
  # Ayuda con el problema del gradiente que tiende a desaparecer
  # Dropout para reducir el sobreajuste (overfitting). Nos ayuda a generalizar.
  nn.Linear(input_size, 32),
  nn.BatchNorm1d(32),
  nn.ReLU(),
  nn.Dropout(dropout_rate)
)
```

```
# Para nuestra imágenes como no tiene buena calidad ni tenemos muchos datos,
# vamos a provechar ahora con la arquitectura de ResNet-50 que es más profunda
# Cagamos el modelo pre-entrenado
resnet50 = torchvision.models.resnet50(weights=models.ResNet50_Weights.DEFAULT)
```

```python
# Red final combinando la de features y la de imagenes y con el clasificador final
# La salida de features_fc = 32.
# La salida de images_cnn = 2048 deberia de ser porque es lo que saca ResNet50.
final_input_size = 32 + cnn_output_dim

self.engagement_classifier = nn.Sequential(
    nn.Linear(final_input_size, 64),
    nn.BatchNorm1d(64),
    nn.ReLU(),
    nn.Dropout(dropout_rate),

    nn.Linear(64, 32),
    nn.BatchNorm1d(32),
    nn.ReLU(),
    nn.Dropout(dropout_rate),

    nn.Linear(32, num_classes) # Capa de salida para 3 clases
)
```

4.- Entrenamiento y Optimización

- ✓ Entrenamiento a las diferentes arquitecturas.
- ✓ De ser necesario creación de una cuarta basándonos en los resultados.
- ✓ Optimización de algunos hiperparámetros.
- ✓ Pruebas cambiando número de épocas o bach size.

Trabajaremos con:

- ✓ Tamaño del batc -----> batch_size = 32
- ✓ Tasa de aprendizaje -----> learning_rate = 0.001
- ✓ Épocas -----> num_epochs = 10
- ✓ Regularización L2 -----> weight_decay = 1e-4
- ✓ Espera para Early Stopping -----> patience = 5
- ✓ Cantidad de capas a congelar -----> freeze_layers = 7
- ✓ Proporción de neuronas se no se van a usar, para reducir el overfitting -----> dropout_rate = 0.3

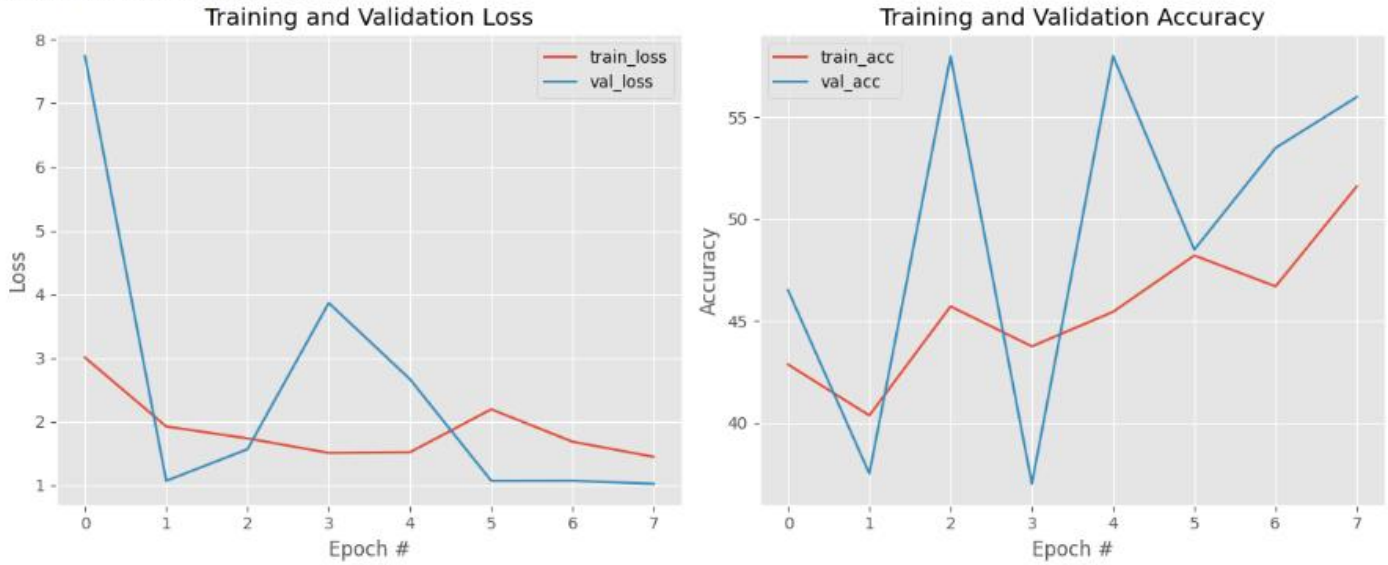
Todos los valores anteriores fueron escogidos basandonos en las recomendaciones de clase e investigación.

- ✓ Función de Perdida -----> nn.CrossEntropyLoss
  - ➢ Utilizamos CrossEntropyLoss dado que es la más usada por sus buenos resultados y porque con ella nos ahorramos usar Softmax.
- ✓ Optimizador -----> Adam
  - ➢ Porque es uno de los más usados con mejor resultados para ajustar los pesos.Es muy eficiente, ligero y converge más rápido que SGD.
- ✓ Programador de la tasa de aprendizaje -----> ReduceLROnPlateau
  - ➢ Porque reduce automáticamente la tasa de aprendizaje cuando una métrica de rendimiento deja de mejorar, basandonos en el limite de epocas que le demos.

4.1 Primer modelo (NetEngagementPOI)

```
Vamos a entrenar
Epoch 1/10, Train Loss: 3.0058, Train Acc: 42.86%, Val Loss: 7.7392, Val Acc: 46.50%
  --> Mejor modelo guardado en 'best_models/best_model_2025-06-23_10-05-24.pth' con Val Acc: 46.5000
Epoch 2/10, Train Loss: 1.9233, Train Acc: 40.36%, Val Loss: 1.0692, Val Acc: 37.50%
Epoch 3/10, Train Loss: 1.7361, Train Acc: 45.71%, Val Loss: 1.5660, Val Acc: 58.00%
  --> Mejor modelo guardado en 'best_models/best_model_2025-06-23_10-05-24.pth' con Val Acc: 58.0000
Epoch 4/10, Train Loss: 1.5078, Train Acc: 43.75%, Val Loss: 3.8636, Val Acc: 37.00%
Epoch 5/10, Train Loss: 1.5188, Train Acc: 45.45%, Val Loss: 2.6659, Val Acc: 58.00%
Epoch 6/10, Train Loss: 2.1929, Train Acc: 48.21%, Val Loss: 1.0697, Val Acc: 48.50%
Epoch 7/10, Train Loss: 1.6843, Train Acc: 46.70%, Val Loss: 1.0714, Val Acc: 53.50%
Epoch 8/10, Train Loss: 1.4462, Train Acc: 51.61%, Val Loss: 1.0231, Val Acc: 56.00%
  ¡Pérdida de validación no mejoró por 5 épocas consecutivas.
El mejor accuracy de validación es: 58.00%

Tiempo de entrenamiento: 123.62 segundos
```



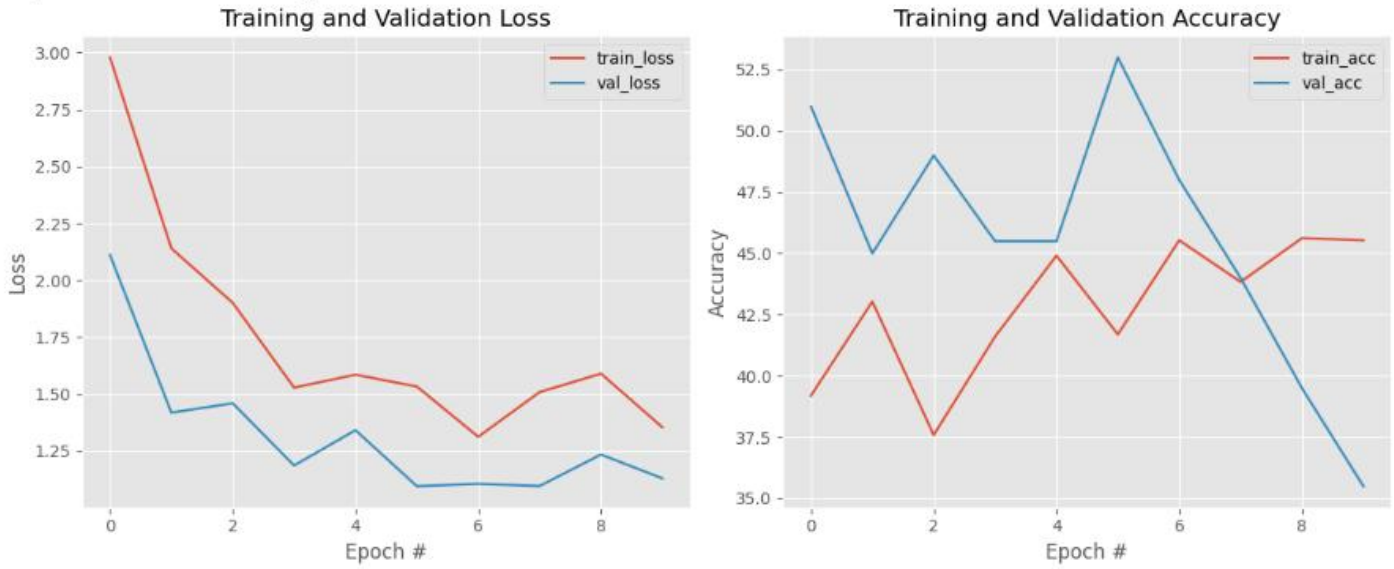
Podemos observar:

- ✓ La perdida en validación cae, oscila bastante, esto nos dice que es inestable y termina abajo de la de entrenamiento (la que va decreciendo constantemente)
- ✓ Nuestro acc de entrenamiento va hacia la alza con algunas oscilaciones, mientras que en entrenamiento tiene varias oscilaciones (muy volatil) y termina subiendo arriba de la de entrenamiento.
- ✓ El último valor Val Acc: 56.00%, esta por arriba de lo que adivinaríamos al azar (33.33%)
- ✓ El mejor accuracy de validación es: 58.00%
- ✓ El tomó 123.62 segundos.
- ✓ No completó el total de las épocas porque se aplico Early Stopping.

4.2 Segundo modelo (NetEngagementPOIProfunda)

```
Vamos a entrenar
Epoch 1/10, Train Loss: 2.9789, Train Acc: 39.20%, Val Loss: 2.1115, Val Acc: 51.00%
  --> Mejor modelo guardado en 'best_models/best_model_2025-06-23_10-10-16.pth' con Val Acc: 51.0000
Epoch 2/10, Train Loss: 2.1403, Train Acc: 43.04%, Val Loss: 1.4187, Val Acc: 45.00%
Epoch 3/10, Train Loss: 1.9010, Train Acc: 37.59%, Val Loss: 1.4596, Val Acc: 49.00%
Epoch 4/10, Train Loss: 1.5286, Train Acc: 41.61%, Val Loss: 1.1864, Val Acc: 45.50%
Epoch 5/10, Train Loss: 1.5853, Train Acc: 44.91%, Val Loss: 1.3416, Val Acc: 45.50%
Epoch 6/10, Train Loss: 1.5333, Train Acc: 41.70%, Val Loss: 1.0955, Val Acc: 53.00%
  --> Mejor modelo guardado en 'best_models/best_model_2025-06-23_10-10-16.pth' con Val Acc: 53.0000
Epoch 7/10, Train Loss: 1.3125, Train Acc: 45.54%, Val Loss: 1.1058, Val Acc: 48.00%
Epoch 8/10, Train Loss: 1.5088, Train Acc: 43.84%, Val Loss: 1.0963, Val Acc: 44.00%
Epoch 9/10, Train Loss: 1.5899, Train Acc: 45.62%, Val Loss: 1.2346, Val Acc: 39.50%
Epoch 10/10, Train Loss: 1.3544, Train Acc: 45.54%, Val Loss: 1.1299, Val Acc: 35.50%
El mejor accuracy de validación es: 53.00%

Tiempo de entrenamiento: 243.17 segundos
```



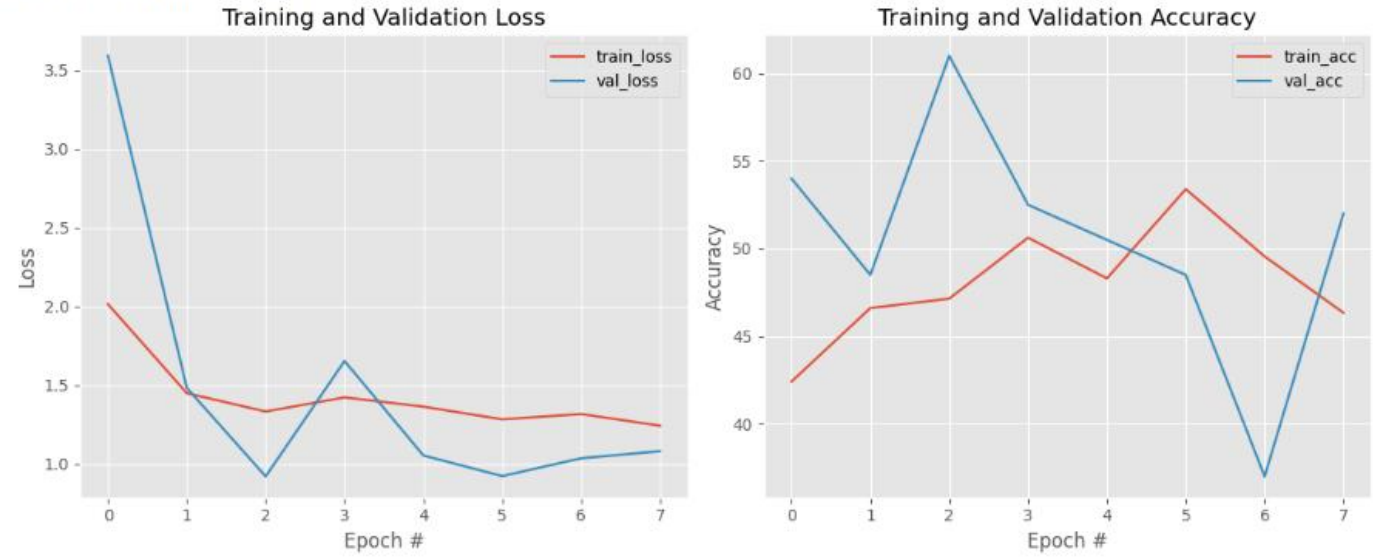
Podemos observar:

- ✓ La perdida en entrenamiento cae consistentemente y la de validación también. Parece que el modelo esta aprendiendo bien.
- ✓ Nuestro acc de entrenamiento se va a la alza, mientras que en entrenamiento empiza alto y va bajando muchisimo. Existe overfitting.
- ✓ Nuestro modelo no generaliza bien.
- ✓ El último valor Val Acc: 35.50%, un poco arriba de que adivinaramos al azar (33.33%), mucho peor que la anterior.
- ✓ El mejor accuracy de validación es: 53.00%, peor que el anterior.
- ✓ Le tomó 243.17 segundos, mucho mayor que la anterior pero era de esperarse dado que la red es mas profunda.
- ✓ Completó las épocas.

4.3 Tercer modelo (NetEngagementPOIMix)

```
Vamos a entrenar
Epoch 1/10, Train Loss: 2.0144, Train Acc: 42.41%, Val Loss: 3.5889, Val Acc: 54.00%
  --> Mejor modelo guardado en 'best_models/best_model_2025-06-23_09-58-04.pth' con Val Acc: 54.0000
Epoch 2/10, Train Loss: 1.4499, Train Acc: 46.61%, Val Loss: 1.4834, Val Acc: 48.50%
Epoch 3/10, Train Loss: 1.3337, Train Acc: 47.14%, Val Loss: 0.9222, Val Acc: 61.00%
  --> Mejor modelo guardado en 'best_models/best_model_2025-06-23_09-58-04.pth' con Val Acc: 61.0000
Epoch 4/10, Train Loss: 1.4232, Train Acc: 50.62%, Val Loss: 1.6540, Val Acc: 52.50%
Epoch 5/10, Train Loss: 1.3646, Train Acc: 48.30%, Val Loss: 1.0550, Val Acc: 50.50%
Epoch 6/10, Train Loss: 1.2839, Train Acc: 53.39%, Val Loss: 0.9243, Val Acc: 48.50%
Epoch 7/10, Train Loss: 1.3177, Train Acc: 49.55%, Val Loss: 1.0373, Val Acc: 37.00%
Epoch 8/10, Train Loss: 1.2440, Train Acc: 46.34%, Val Loss: 1.0821, Val Acc: 52.00%
  ¡Pérdida de validación no mejoró por 5 épocas consecutivas.
El mejor accuracy de validación es: 61.00%

Tiempo de entrenamiento: 197.05 segundos
```



Podemos observar:

- ✓ La perdida en entrenamiento cae consistente, y la de validación también con alguna oscilación pero termina en un valor mas bajo que la de entrenamiento.
- ✓ Nuestro acc de entrenamiento parece que subia pero de repente comenzo a bajar, mientras que en validación muestra oscilaciones importantes siendo inestable.
- ✓ El último valor Val Acc: 52.00%, por encima del azar (33.33%), el mejor que el anterior pero no que la primera red.
- ✓ El mejor accuracy de validación es: 61.00%, mejor que los anteriores.
- ✓ Le tomó 197.05 segundos. Entre los dos anteriores.

4.3 Creación de Cuarto modelo (NetEngagementPOIFinal)

Basandonos en los resultados anteriores, sobre todo en la mejor precisión de validación alcanzada antes del overfitting sería la primera red "NetEngagementPOI".

Voy a intentar modificar la primera para ver si mejora:

- ✓ Quitar capas en el clasificador final
- ✓ En la red para features vamos a hacer la ultima más estrecha.

Este modelo incluira:

- ✓ Red para features full connected (3 capas con salidas 64, 32 y 16)
- ✓ Red para imágenes trabajaremos con ResNet-18
- ✓ Para la red que las combina full connected (1 capa con salida 64)

```
# Para nuestra features trabajaremos con capas full connected
self.features_fc = nn.Sequential(

  # Primera Capa full connected
  # BatchNorm1d nos ayuda normalizar la salida de la capa antrior y acelerar el entrenamiento.
  # Función de activación Relu dado que para en general funicona muy bien para clasificación.
  # Ayuda con el problema del gradiente que tiende a desaparecer
  # Dropout para reducir el sobreajuste (overfitting). Nos ayuda a generalizar.
  nn.Linear(input_size, 64),
  nn.BatchNorm1d(64),
  nn.ReLU(),
  nn.Dropout(dropout_rate),

  # Segunda Capa full connected
  nn.Linear(64, 32),
  nn.BatchNorm1d(32),
  nn.ReLU(),
  nn.Dropout(dropout_rate),

  # Tercera Capa full connected
  nn.Linear(32, 16),
  nn.BatchNorm1d(16),
  nn.ReLU(),
  nn.Dropout(dropout_rate)

)
```

```
# Para nuestra imágenes como no tiene buena calidad ni tenemos muchos datos, vamos a provechar la arquitectura de ResNet-18
# Cagamos el modelo pre-entrenado
resnet18 = torchvision.models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
```

```
# Red final combinando la de features y la de imagenes y con el clasificador final
# La salida de features_fc = 16.
# La salida de images_cnn = 512 deberia de ser porque es lo que saca ResNet18.
final_input_size = 16 + cnn_output_dim

self.engagement_classifier = nn.Sequential(
    nn.Linear(final_input_size, 64),
    nn.BatchNorm1d(64),
    nn.ReLU(),
    nn.Dropout(dropout_rate),

    nn.Linear(64, num_classes) # Capa de salida para 3 clases
)
```

Los resultados obtenidos son:

```
Vamos a entrenar
Epoch 1/10, Train Loss: 2.3364, Train Acc: 48.57%, Val Loss: 3.7813, Val Acc: 54.00%
  --> Mejor modelo guardado en 'best_models/best_model_2025-06-22_17-10-47.pth' con Val Acc: 54.0000
Epoch 2/10, Train Loss: 1.3767, Train Acc: 51.07%, Val Loss: 4.0380, Val Acc: 57.50%
  --> Mejor modelo guardado en 'best_models/best_model_2025-06-22_17-10-47.pth' con Val Acc: 57.5000
Epoch 3/10, Train Loss: 0.9763, Train Acc: 54.11%, Val Loss: 1.0336, Val Acc: 63.00%
  --> Mejor modelo guardado en 'best_models/best_model_2025-06-22_17-10-47.pth' con Val Acc: 63.0000
Epoch 4/10, Train Loss: 0.9046, Train Acc: 59.02%, Val Loss: 0.8585, Val Acc: 59.00%
Epoch 5/10, Train Loss: 0.9298, Train Acc: 57.23%, Val Loss: 1.0568, Val Acc: 57.50%
Epoch 6/10, Train Loss: 0.9795, Train Acc: 58.84%, Val Loss: 0.7733, Val Acc: 59.00%
Epoch 7/10, Train Loss: 0.9354, Train Acc: 60.36%, Val Loss: 1.2586, Val Acc: 56.50%
Epoch 8/10, Train Loss: 1.1041, Train Acc: 58.39%, Val Loss: 1.0578, Val Acc: 50.00%
  ¡Pérdida de validación no mejoró por 5 épocas consecutivas.
El mejor accuracy de validación es: 63.00%

Tiempo de entrenamiento: 117.25 segundos
```

Podemos observar:

- ✓ Podemos observar que la perdida en entrenamiento disminuye de manera constante.
- ✓ La perdida en validación empieza alta, pero rápido cae y termina mas abajo que la de entrenamiento. Tiene algunas oscilaciones pero en general la tendencia es a la baja.
- ✓ La precisión en entrenamiento muestra una tendencia creciente y medio estable mientras que la de validación alcanza un punto muy alto y luego tiende a bajar.
- ✓ El último valor Val Acc: 50.00%, el mejor de todos.
- ✓ El mejor accuracy de validación es: 63.00%, el mejor de todos.
- ✓ Le tomó 117.25 segundos, es más rápido que todos

Nos quedamos con este modelo porque:

- ✓ Tiene el mejor rendimiento de todas.
- ✓ Es la más rápida.
- ✓ La perdida en validación tiene tendencia a la baja.

Nuestro modelo queda:

```
NetEngagementPOIFinal(
  (features_fc): Sequential(
    (0): Linear(in_features=22, out_features=64, bias=True)
    (1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
```

```
  (3): Dropout(p=0.4, inplace=False)
  (4): Linear(in_features=64, out_features=32, bias=True)
  (5): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (6): ReLU()
  (7): Dropout(p=0.4, inplace=False)
  (8): Linear(in_features=32, out_features=16, bias=True)
  (9): BatchNorm1d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (10): ReLU()
  (11): Dropout(p=0.4, inplace=False)
)
(images_cnn): Sequential(
  (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (5): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   )
  )
  (6): Sequential(
   (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
     (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
     (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
   )
   (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   )
  )
  (7): Sequential(
   (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
     (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
     (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
   )
   (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   )
  )
  (8): AdaptiveAvgPool2d(output_size=(1, 1))
  (9): Flatten(start_dim=1, end_dim=-1)
 )
 (engagement_classifier): Sequential(
  (0): Linear(in_features=528, out_features=64, bias=True)
```

```
  (1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU()
  (3): Dropout(p=0.4, inplace=False)
  (4): Linear(in_features=64, out_features=3, bias=True)
 )
)
```

Nota: Podemos observarla a detalle en la imagen model_graph_final.png

4.4 Optimización

Vamos a Optimizar los hiper-parámetros con Optuna. Queremos optimizar 'Dropout Rate' y 'Freeze Layers'.

Los valores a evaluar son:

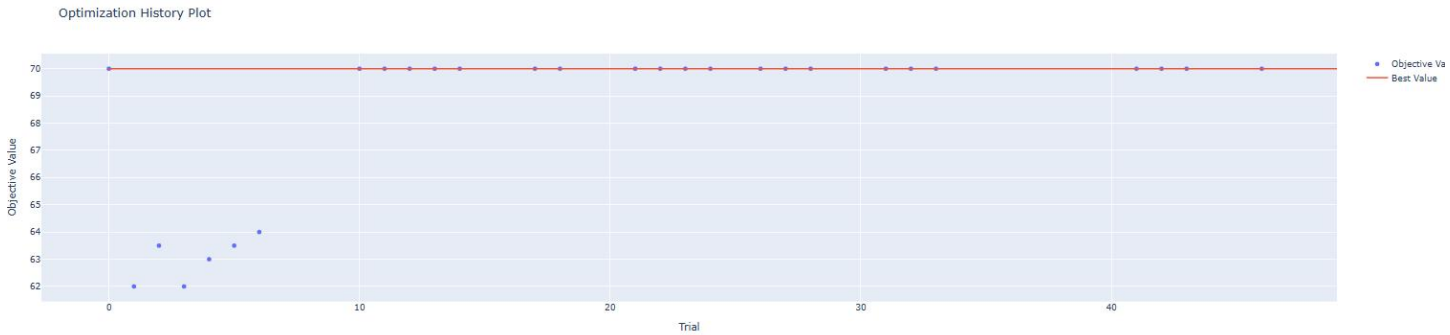
- ✓ "dropout_rate" = [0.1, 0.2, 0.3, 0.4, 0.5]
- ✓ "freeze_layers" = [5, 6, 7, 8, 9]

Los mejores trials fueron:

Número de trial	Valor
#0	70.0000
#10	70.0000
#11	70.0000
#12	70.0000
#13	70.0000

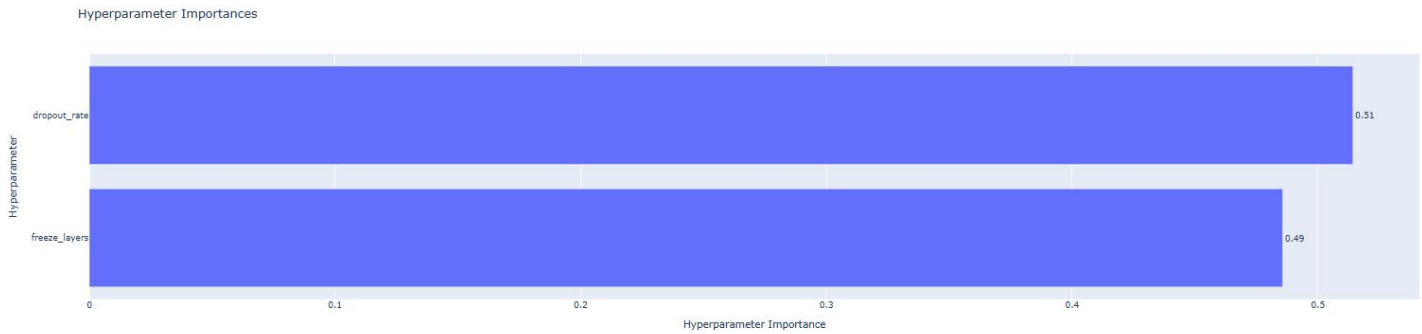
Obtuvimos las siguientes gráficas:

<u>Progreso de la optimización de los hiperparámetros a lo largo del tiempo</u>



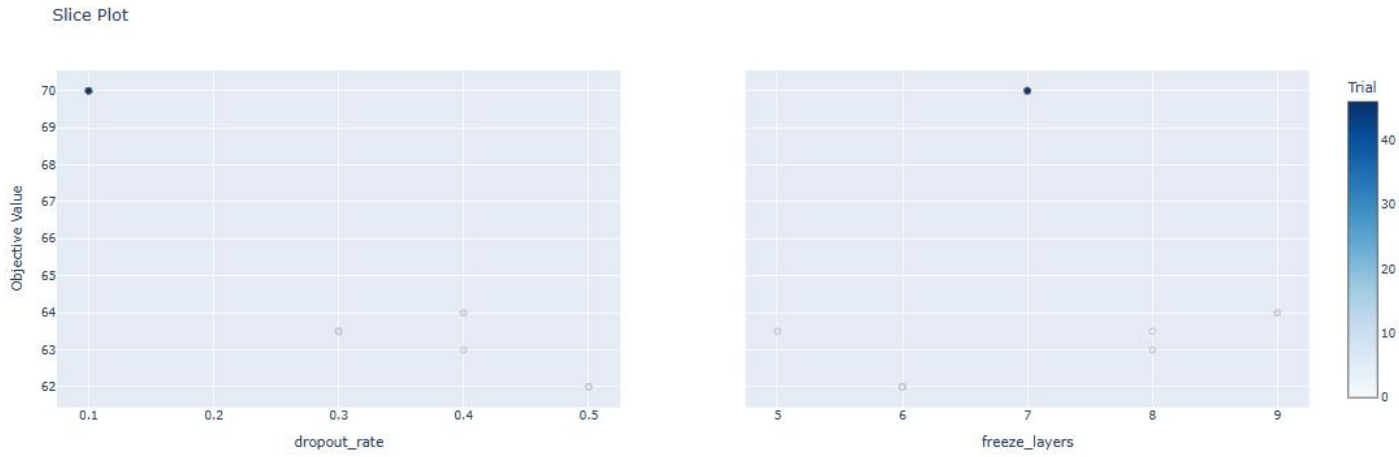
El proceso de optimización ha convergido muy rápidamente. Optuna encontró un valor de 70 (o muy cercano) casi de inmediato y no ha logrado superarlo en los trials posteriores.

<u>Importancia de los hiperparámetros</u>



Podemos observar que ambos parámetros tienen la misma importancia. Ambas tienen un impacto significativo y casi idéntico en el rendimiento del modelo.

Relación de los hiperparámetros con el valor objetivo



Para el espacio de búsqueda que definimos, las mejores combinaciones de hiperparámetros son: dropout_rate aprox. 0.1 y freeze_layers igual a 7.

Podriamos volver a ejecutar Optuna modificando el rango dropout_rate con valores alrededor de 0.1, el de freeze_layers no lo incluiremos porque el resultado obtenido esta justo en el centro del rango que abarcamos.

También vamos a incrementar un poco el número de epocas a 15, no es un gran cambio porque en validación nuestra precisión tenía tendencia a ir a la baja.

Y el batch_size a 16 para ayudar a nuestra GPU, ver si nos ayuda a generalizar mejor.

Los valores a evaluar son:

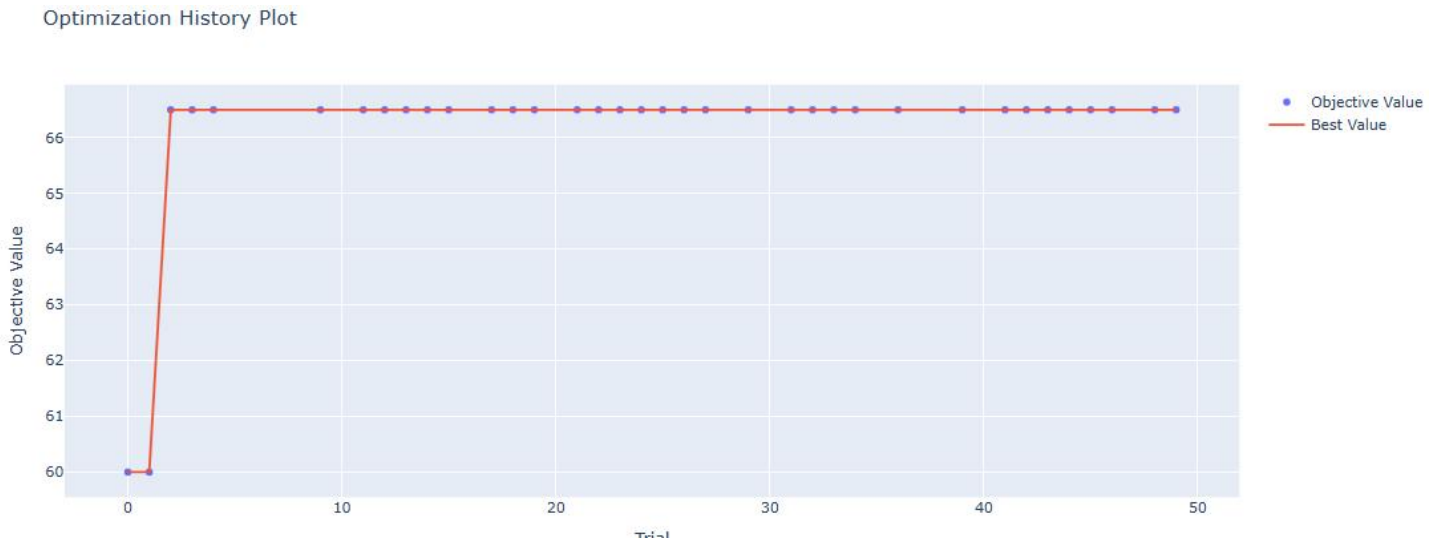
- ✓ "dropout_rate" = [0.05, 0.1, 0.15]

Los mejores trials fueron:

```
Mejores trials:
Número de trial Valor
#2              66.5000
#3              66.5000
#4              66.5000
#9              66.5000
#11             66.5000
```

Obtuvimos las siguientes gráficas:

Progreso de la optimización de los hiperparámetros a lo largo del tiempo



Comparado con el estudio anterior, tardo un poco más en encontrar el valor óptimo y este no llego a ser mejor ni igual al anterior.

Basándonos en los resultados obtenidos en este segundo intento con Optuna, nos quedamos con el mejor modelo de la primera vez que ejecutamos la optimización de hiperparámetros con Optuna.

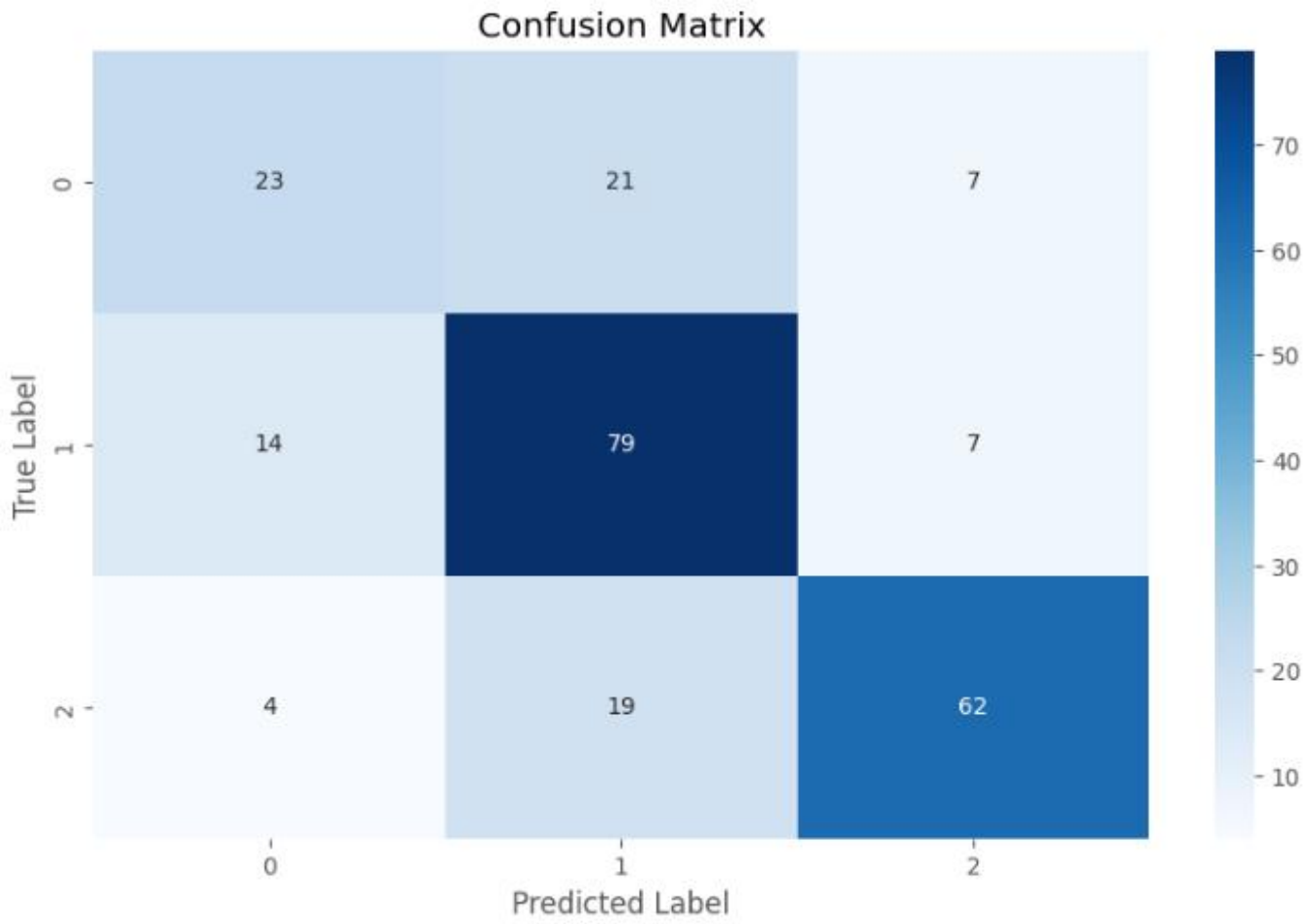
5. - Evaluación y Análisis

Vamos a cargar el mejor modelo obtenido así como los valores de los hiperparmetros.

```
Best trial accuracy: 70.0
Best hyperparameters: {'dropout_rate': 0.1, 'freeze_layers': 7}
```

Despues evaluar el rendimiento (ver que tan bien generaliza) de nuestro modelo con el conjunto de datos de prueba (test).

Nuestros resultados de la evaluación es:



Classification Report:

	precision	recall	f1-score	support
Bajo	0.56	0.45	0.50	51
Medio	0.66	0.79	0.72	100
Alto	0.82	0.73	0.77	85
accuracy			0.69	236
macro avg	0.68	0.66	0.66	236
weighted avg	0.70	0.69	0.69	236

5.1 Matriz de Confusión y Reporte de Clasificación

Las filas representan las clases reales mientras que las columnas las predichas. Podemos observar el detalle de como se distribuyen los aciertos y errores:

- ✓ **23 registros que realmente eran 0 fueron predichos como 0**
- ✓ 21 registros que realmente eran 0 fueron predichos como 1
- ✓ 7 registros que realmente eran 0 fueron predichos como 2

Total de clase 0 reales = 51

- ✓ 14 registros que realmente eran 1 fueron predichos como 0

- ✓ **79 registros que realmente eran 1 fueron predichos como 1**
- ✓ 7 registros que realmente eran 1 fueron predichos como 2

Total de clase 1 reales = 100

- ✓ 4 registros que realmente eran 2 fueron predichos como 0
- ✓ 19 registros que realmente eran 2 fueron predichos como 1
- ✓ **62 registros que realmente eran 2 fueron predichos como 2**

Total de clase 2 reales = 85

Observamos que tienen mejor rendimiento las clases 1 y 2:

- ✓ **Clase 1 con un 79% de recall**, es decir que por cada 100 instancias de clase 1, 79 fueron encontradas correctamente.
- ✓ **Clase 2 con un 73% de recal**l, es decir que por cada 100 instancias de clase 2 73 fueron encontradas correctamente.
- ✓ En cambio la **Clase 0 tuvo más problemas**, muestra un 45% en recall, dejando de identificar correctamente casi la mitad.

Esto también lo podemos confirmar con la precisión que confirma que la clase 0 es en la que más problemas tuvo (56% de probabilidades de estar en lo correcto). Aunque a la clase 1 también le da un valor más bajo que en el recall, con un 66% de estar en lo correcto.

Observando el f1-score, si es alto nos indica que el model tiene buena precisión y buen recall. Con esta métrica confirmamos que la clase 0 es quien presenta mayores problemas.

69% es la precisión general del modelo en el conjunto de datos de test, es decir que el modelo acierta el 69% de las predicciones.

En general podemos decir que nuestro modelo tiene un rendimiento general aceptable. Siendo más eficaz para las clases 1 y 2.

5.2 Errores más representativos

Para la clase 0:

- ✓ Cuenta con 29 Falsos Negativos en general, el error más representativo es con la clase 1 (21 casos).

  Esto puede suceder porque a los mejor las caracteristicas que nos definen a la clase 0 sean poco evidentes o comparta algunas con la clase 1 y por esta razón el modelo las confunda.

- ✓ Cuenta con 18 Falsos Positivos en general, el error más representativo es con la clase 1 (14 casos).

  Con esto reafirmamos que el modelo no solo falla al identificar la clase 0 reales, sino que también clasifica erróneamente las clases 1 en clase 0.

Para la clase 1:

- ✓ Cuenta con 21 Falsos Negativos en general, el error más representativo es con la clase 0 (14 casos).

  Esto puede sucede el modelo confunde la clase 1 con la clase 0.

- ✓ Cuenta con 40 Falsos Positivos en general, el error más representativose encuentra en ambas clases dado que cuenta con 21 casos para la clase 0 y 19 para la clase 2.

Para la clase 2:

- ✓ Cuenta con 23 Falsos Negativos en general, el error más representativo es con la clase 1 (19 casos).

  Donde la instancia era clase 2 y el modelo la predijo como clase 1.

- ✓ Cuenta con 14 Falsos Positivos en general, el error más representativo se encuentra en ambas clases dado que cuenta con 7 casos para la clase 0 y 7 para la clase 1.

Podemos observar que la mayoría de confusiones corren entre las clases contiguas, es decir, clase 0 con clase 1 o clase 1 con clase 2. Es menos común la confusión entre clase 0 y clase 2, pero si es más grave. Todo esto nos dice que las fronteras de decisión entre las clases no están muy bien definidas.

5.3 Propuestas de mejoras futuras.

Si tuviera más tiempo:

- Primero realizaría un rediseño de mi código para que fuera más fácil realizar cambios en los parametros para realizar más pruebas con batch size, número de epocas, mayores hiperparámetros en Optuna.

- Probaría a realizar una diferente distribución (porcentajes) entre los conjuntos de datos.

- Realizaría gráficas de Mapas de calor para ver las variables más relevantes para predecir, con Grad-CAM.

- Trabajaría para obtener la importancia de las variables, aplicando Model Agnostic - SHAP/LIME (necesitamos de los porque nuestro modelo contiene red CNN y red full connected) (nunca he trabajado con estos modelos así que necesitaría más tiempo para estudiarlos.)

- ✓ SHAP (SHapley Additive exPlanations): Asigna un valor de "importancia" a cada característica para una predicción individual, indicando cómo cada característica contribuye a la predicción frente a la predicción base. Esto nos ayuda a ver porque un punto de interés fue clasificado en una clase errónea.

- ✓ LIME (Local Interpretable Model-agnostic Explanations): nos explica las predicciones de cualquier clasificador o regresor de "caja negra" . Nos puede decir qué características fueron importantes.