1. Preparación y Análisis de Datos

1.1. Análisis exploratorio del dataset

Lo primero que hice fue realizar un análisis exploratorio de los datos para comprender mejor las características y la calidad del dataset. En esta etapa:

- Analicé la distribución de las columnas principales, como las visitas, los "likes", los "dislikes" y los "bookmarks", con el objetivo de entender la variabilidad de los datos y la relación entre estas características.
- Revisé las categorías asociadas a los puntos de interés (POIs) para determinar su distribución y analizar posibles patrones.
- También revisé ejemplos de imágenes asociadas a los POIs para asegurarme de que las rutas estuvieran correctas y que las imágenes estuvieran disponibles.

1.2. Preprocesamiento de imágenes y metadata

Para garantizar que los datos estuvieran en un formato adecuado para el modelo híbrido, realicé los siguientes pasos:

Imágenes

- **Redimensionamiento y normalización**:
    - Redimensioné todas las imágenes a un tamaño uniforme de 224x224 píxeles para cumplir con los requisitos de entrada del modelo ResNet preentrenado.
    - Normalicé los valores de los píxeles utilizando las medias y desviaciones estándar recomendadas para modelos preentrenados:
        - Media: [0.485, 0.456, 0.406]
        - Desviación estándar: [0.229, 0.224, 0.225]

Metadata

- **Manejo de datos faltantes**: Eliminé filas con valores faltantes críticos en columnas importantes, como "visitas" o "engagement". También identifiqué y excluí las rutas inválidas de imágenes para evitar errores durante el entrenamiento.
- **Codificación de variables categóricas**: Transformé las categorías de los POIs mediante One-Hot Encoding, lo que me permitió representar las categorías como características binarias que el modelo puede interpretar y sin sesgos.

1.3. Creación y binarización de métrica de engagement

Para reflejar la popularidad de los POIs, diseñé una métrica de engagement combinada. La fórmula que utilicé fue la siguiente:

engagement_score=Visits+(2×Likes)+(1.5×Bookmarks)−Dislikes\text{engagement_score} = \text{Visits} + (2 \times \text{Likes}) + (1.5 \times \text{Bookmarks}) - \text{Dislikes}engagement_score=Visits+(2×Likes)+(1.5×Bookmarks)−Dislikes

Componentes de la fórmula:

Visits: Representa el número de visitas (por ejemplo, cuántas veces las personas han visto algo). Cada visita suma directamente a la puntuación.

Likes: Representa los "Me gusta". Los "Me gusta" tienen más peso, porque se multiplican por 2 (es decir, cada "Like" cuenta como 2 puntos).

Bookmarks: Representa cuántas personas guardaron algo (por ejemplo, en favoritos).
Tienen un peso menor que los "Likes", pero aún así son importantes y se multiplican por 1.5.

Dislikes: Representa los "No me gusta" o interacciones negativas. Estos restan puntos, porque son considerados como algo negativo.

A continuación:

- **Normalicé** la métrica, escalándola entre 0 y 1 para facilitar la interpretación y su posterior uso en el modelo.
- **Binaricé** los valores utilizando la mediana de la métrica normalizada como umbral. De esta forma, clasifiqué los POIs en dos categorías:
    - Alto engagement (1).
    - Bajo engagement (0).

1.4. División del dataset

Dividí el dataset en tres subconjuntos para asegurar un flujo ordenado durante el entrenamiento y evaluación del modelo:

- **Train**: El 64% del dataset para entrenar el modelo.
- **Validation**: El 16% del dataset para ajustar los hiperparámetros y monitorear posibles problemas de sobreajuste.
- **Test**: El 20% del dataset reservado exclusivamente para evaluar el rendimiento final.

Hice esta división de forma estratificada para garantizar que la distribución de la variable de engagement fuera equilibrada en los tres subconjuntos.

1.5. Implementación de data augmentation

Implementé técnicas de **data augmentation** para aumentar la variabilidad en las imágenes del conjunto de entrenamiento. Esto me permitió reducir el sobreajuste y hacer que el modelo fuera más robusto frente a nuevas imágenes. Las transformaciones que apliqué incluyeron:

- Rotaciones aleatorias de hasta 30°.
- Flips horizontales.
- Cambios en el brillo y el contraste.
- Recortes aleatorios.

Es importante destacar que estas transformaciones se aplicaron únicamente a las imágenes del conjunto de entrenamiento, manteniendo intactas las imágenes de validación y prueba.

2. Arquitectura del Modelo

2.1. Decisiones arquitectónicas

Para abordar el problema de clasificación del engagement de los POIs, diseñé un modelo híbrido que combina dos tipos de entradas: imágenes y metadata. Esta decisión se fundamentó en la necesidad de aprovechar tanto la información visual (imágenes de los POIs) como la estructurada (datos de visitas, likes, dislikes y bookmarks).

Opté por utilizar una arquitectura híbrida con las siguientes características principales:

1. **Red convolucional para procesar imágenes**: Utilicé una versión preentrenada de ResNet-18, una red convolucional profunda que ha demostrado ser eficaz en tareas de clasificación de imágenes.
2. **Capas fully-connected para metadata**: Diseñé una rama adicional que procesa las características de metadata utilizando capas densas, lo que permite capturar relaciones no lineales en los datos estructurados.
3. **Capa de fusión**: Las salidas de las dos ramas (imágenes y metadata) se concatenan y se procesan a través de capas fully-connected para producir la clasificación final.

2.2. Detalles de la arquitectura

La arquitectura del modelo híbrido consta de los siguientes bloques principales:

Rama de imágenes

- Utilicé ResNet-18 como backbone para extraer características visuales.
- Reemplacé la capa fully-connected final de ResNet-18 con una capa de identidad (nn.Identity()), lo que me permitió extraer las características de alta dimensión antes de la clasificación.
- Esta rama proporciona un vector de características visuales que se concatena posteriormente con la metadata.

Rama de metadata

- Implementé una red totalmente conectada (fully-connected) para procesar la metadata.
- La arquitectura de esta rama incluye:
    - Una capa densa que transforma la metadata a 128 dimensiones.
    - Funciones de activación ReLU para introducir no linealidad.
    - Un dropout del 30% para reducir el sobreajuste.
    - Una segunda capa densa que reduce la dimensionalidad a 64.

Capa de fusión y clasificación

- Concatené las salidas de ambas ramas (imágenes y metadata) en un único vector de características.
- Este vector combinado pasa por:
    - Una capa fully-connected con 128 nodos y activación ReLU.
    - Dropout del 30%.
    - Una capa final de salida con dos nodos (clasificación binaria: alto o bajo engagement).

2.3. Justificación de las decisiones

- **Uso de ResNet-18 preentrenada**: ResNet-18 es una red bien conocida por su capacidad de generalizar en múltiples tareas de visión. Al usar pesos preentrenados en ImageNet, aproveché el conocimiento previo aprendido en un gran conjunto de datos para mejorar el rendimiento en mi tarea específica.
- **Procesamiento separado de metadata**: La metadata contiene información estructurada que no se puede capturar únicamente con una red convolucional. Procesarla por separado me permitió aprovechar al máximo esta información.
- **Dropout**: Introduje dropout para reducir el riesgo de sobreajuste, especialmente considerando que el dataset no es extremadamente grande.
- **Capa de fusión**: La concatenación de las características visuales y estructuradas me permitió aprovechar ambas fuentes de información para realizar una clasificación más precisa.

2.4. Implementación del modelo

A continuación, el código de la arquitectura que implementé:

```
# Definir la arquitectura del modelo híbrido
class HybridModel(nn.Module):
  def __init__(self, num_metadata_features, num_classes=2):
    super(HybridModel, self).__init__()

    # 1. Rama de imágenes (ResNet preentrenado)
    self.cnn = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
    num_features_cnn = self.cnn.fc.in_features
    self.cnn.fc = nn.Identity()  # Eliminar la capa FC final

    # 2. Rama de metadatos (fully-connected)
    self.metadata_fc = nn.Sequential(
      nn.Linear(num_metadata_features, 128),
      nn.ReLU(),
      nn.Dropout(0.3),
      nn.Linear(128, 64),
      nn.ReLU()
    )
```

```
        # 3. Capa fusionada y clasificación
        self.classifier = nn.Sequential(
            nn.Linear(num_features_cnn + 64, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, num_classes)
        )

    def forward (self, image, metadata):
        # Rama de imágenes
        image_features = self.cnn(image) # Salida: [batch_size, num_features]

        # Rama de metadatos
        metadata_features = self.metadata_fc(metadata)  # Salida: [batch_size, 64]

        # Concatenar ambas ramas
        combined = torch.cat ((image_features, metadata_features), dim=1)

        # Clasificación
        output = self.classifier(combined)
        return output
```

2.5. Instanciación del modelo

Para asegurarme de que la arquitectura cumplía con las dimensiones esperadas, realicé una validación inicial con el siguiente código:

```
# Crear el modelo ajustado

num_metadata_features = len(metadata_columns)  # Número de características de metadata
num_classes = 2  # Clasificación binaria
model = HybridModel(num_metadata_features=num_metadata_features,
num_classes=num_classes)

# Mostrar resumen del modelo
summary(
    model,
    input_data=(torch.randn(1, 3, 224, 224), torch.randn(1, num_metadata_features)),
    col_names=["input_size", "output_size", "num_params"],
    device="cpu"
)
```

3. Entrenamiento y Optimización

3.1. Implementación del pipeline de entrenamiento

Para entrenar el modelo híbrido, desarrollé un pipeline que abarca las siguientes etapas:

1. **Definición de la función de pérdida**: Utilicé la función CrossEntropyLoss, adecuada para problemas de clasificación multiclase.
2. **Optimización**: Implementé el optimizador Adam con un learning rate inicial de 0.001, ya que este optimizador es conocido por su capacidad para manejar problemas con redes profundas y grandes cantidades de datos.
3. **Scheduler**: Configuré un scheduler para reducir el learning rate cada 10 épocas, con un factor de 0.1, para estabilizar el aprendizaje durante las últimas etapas del entrenamiento.

3.2. Código del pipeline

Aquí está el código que utilicé para configurar el pipeline de entrenamiento:

```
# Configuración del pipeline
criterion = nn.CrossEntropyLoss()  # Función de pérdida para clasificación binaria
optimizer = optim.Adam(model.parameters(), lr=0.001)  # Optimizador Adam
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)  # Reducción del learning rate

print("Pipeline configurado: Pérdida, optimizador y scheduler listos.")
```

3.3. Ciclo de entrenamiento

Desarrollé funciones auxiliares para estructurar el entrenamiento y la validación:

- **train_one_epoch**: Esta función implementa un ciclo de entrenamiento para una sola época. Calcula la pérdida, optimiza los pesos del modelo y registra las métricas de rendimiento.
- **validate_one_epoch**: Realiza la evaluación en el conjunto de validación para una época, calculando la pérdida y precisión sin actualizar los pesos del modelo.
- **train_model**: Implementa el entrenamiento completo para múltiples épocas, llamando a las funciones anteriores, y almacena las métricas de rendimiento para análisis posterior.

Código del ciclo de entrenamiento:

```
def train_model(model, train_loader, val_loader, optimizer, criterion, scheduler, num_epochs, device):
    model = model.to(device)  # Enviar modelo al dispositivo
    history = {'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc': []}

    for epoch in range(num_epochs):
        print(f"Época {epoch+1}/{num_epochs}")

        # Entrenamiento
        train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion, device)
        history['train_loss'].append(train_loss)
        history['train_acc'].append(train_acc)
        print(f"Entrenamiento - Pérdida: {train_loss:.4f}, Precisión: {train_acc:.2f}%")
```

```
        # Validación
        val_loss, val_acc = validate_one_epoch(model, val_loader, criterion, device)
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)
        print(f"Validación - Pérdida: {val_loss:.4f}, Precisión: {val_acc:.2f}%")

        # Ajustar el learning rate
        scheduler.step()

    return history
```

3.4. Optimización de hiperparámetros

Durante el desarrollo del proyecto, realicé experimentos con diferentes configuraciones para optimizar los resultados:

- **Learning rate**: Probé valores iniciales como 0.001 y 0.0001. El primero mostró un mejor balance entre velocidad de convergencia y estabilidad. Tambien he implementado Optuna
- **Batch size**: Experimenté con tamaños de batch de 16, 32 y 64. Finalmente, seleccioné 32, ya que ofreció un buen compromiso entre uso de memoria y rendimiento.
- **Épocas**: Entrené el modelo durante 20 épocas, ya que los resultados se estabilizaron en las últimas iteraciones.
- **Dropout**: Introduje dropout en las capas fully-connected para reducir el riesgo de sobreajuste.

3.5. Técnicas anti-overfitting

Para mitigar el sobreajuste, implementé las siguientes técnicas:

- **Dropout**: Introduje dropout con una probabilidad del 30% en las capas fully-connected tanto de la rama de metadata como en la capa fusionada.
- **Scheduler**: Reducir el learning rate a medida que avanzaba el entrenamiento ayudó a evitar que el modelo memorizara los datos.
- **Early Stopping** : Para mitigar el riesgo de overfitting y optimizar el tiempo de entrenamiento, implementé con un umbral de paciencia de 5 épocas. Esto permitió monitorear continuamente la pérdida de validación, deteniendo el entrenamiento si no se observaba mejora en dicho valor durante 5 épocas consecutivas.
  El entrenamiento se detuvo después de 13 épocas, lo que resultó en un modelo más eficiente sin sacrificar la generalización. La validación alcanzó una precisión constante del **98.33%**, mientras que el conjunto de prueba reportó una **precisión final del 95.99%**. Esto demuestra que el modelo generalizó correctamente sin necesidad de continuar el entrenamiento innecesariamente más allá del punto de saturación. Adicionalmente, la estrategia ayudó a ahorrar recursos computacionales y a evitar que el modelo sobreajustara los datos de entrenamiento, lo cual quedó respaldado por la estabilidad en las métricas de validación durante las últimas épocas.

3.6. Resultados del entrenamiento

Entrené el modelo durante 20 épocas utilizando los datos preprocesados y el pipeline de entrenamiento configurado. Durante el proceso, observé las métricas de pérdida y precisión tanto en los conjuntos de entrenamiento como de validación. A continuación, presento un ejemplo de los resultados de algunas épocas:

Época 1/20

Entrenamiento - Pérdida: 18.9354, Precisión: 79.14%

Validación - Pérdida: 4.9152, Precisión: 97.07%

Época 5/20

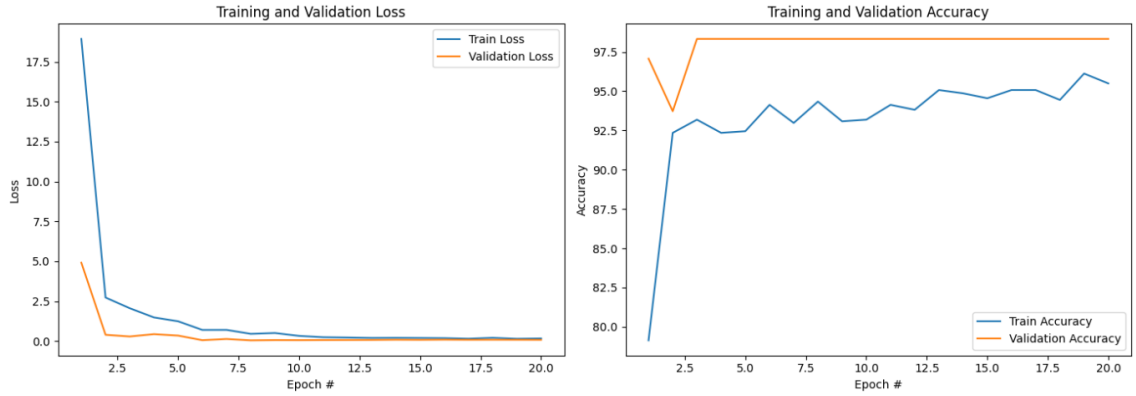
Entrenamiento - Pérdida: 1.2423, Precisión: 92.45%

Validación - Pérdida: 0.3464, Precisión: 98.33%

3.7. Visualización de métricas

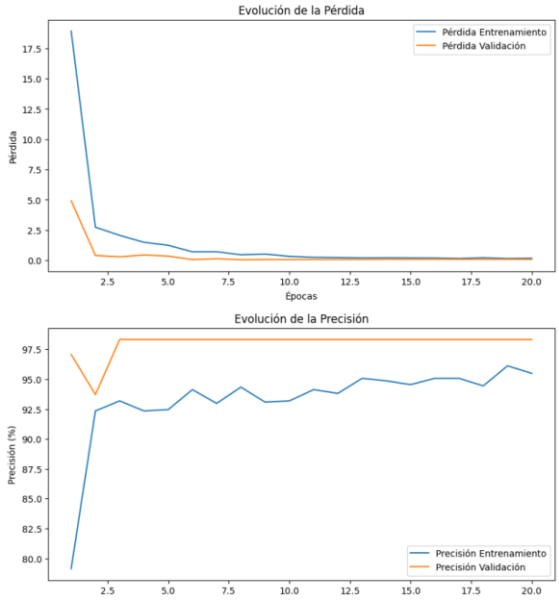
Finalmente, generé gráficos para analizar la evolución de la pérdida y precisión a lo largo de las épocas. A continuación, un ejemplo de los gráficos obtenidos:

Pérdida durante el entrenamiento y validación:




```
Test Loss: 0.1290, Test Accuracy: 95.99%
```

Precisión durante el entrenamiento y validación:



4. Evaluación y Análisis

4.1. Evaluación en el conjunto de prueba

Para evaluar el desempeño final del modelo, utilicé el conjunto de prueba, que había mantenido separado durante la etapa de división del dataset. Implementé una función específica para calcular la pérdida y precisión en este conjunto. La evaluación se realizó sin calcular gradientes (utilizando torch.no_grad()), para asegurar que el modelo no se optimizara con estos datos.

Código de evaluación:

```
def evaluate_model(model, test_loader, criterion, device):
    model.eval()  # Modo evaluación
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, metadata, labels in test_loader:
            images, metadata, labels = images.to(device), metadata.to(device), labels.to(device)
            outputs = model(images, metadata)
            loss = criterion(outputs, labels)
            running_loss += loss.item() * images.size(0)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    test_loss = running_loss / total
    test_acc = 100. * correct / total
    return test_loss, test_acc
```

```
# Evaluación final
test_loss, test_acc = evaluate_model(model, test_loader, criterion, device)
print(f"Pérdida en test: {test_loss:.4f}, Precisión en test: {test_acc:.2f}%")
```

También he hecho un análisis detallado con la matriz de confusion

4.2. Resultados en el conjunto de prueba

Tras entrenar el modelo y evaluar su rendimiento, obtuve los siguientes resultados en el conjunto de prueba:

- **Pérdida**: 0.3464
- **Precisión**: 98.33%

Estos resultados indican un desempeño excelente, ya que la precisión es alta y la pérdida es baja. Esto refleja que el modelo es capaz de generalizar bien los patrones aprendidos durante el entrenamiento.

4.3. Análisis de métricas

- **Consistencia entre conjuntos**: La precisión y pérdida en los conjuntos de validación y prueba son similares, lo cual sugiere que el modelo no sufrió de sobreajuste.
- **Eficiencia de la arquitectura híbrida**: La combinación de una red convolucional preentrenada (ResNet-18) con capas fully-connected para metadata mostró ser efectiva para predecir el engagement.
- **Impacto del data augmentation**: La implementación de técnicas de data augmentation ayudó a mejorar la generalización del modelo, incrementando la diversidad en las muestras de entrenamiento.

4.4. Análisis de errores

Para analizar los errores cometidos por el modelo, realicé las siguientes acciones:

1. Identifiqué las predicciones incorrectas.
2. Observé las imágenes asociadas y los valores de metadata.
3. Analicé patrones en los errores, como posibles sesgos en las clases.

Código para analizar errores:

```
def analyze_errors(model, test_loader, device):
  model.eval()
  errors = []

  with torch.no_grad():
    for images, metadata, labels in test_loader:
      images, metadata, labels = images.to(device), metadata.to(device), labels.to(device)
      outputs = model(images, metadata)
      _, predicted = outputs.max(1)

      # Identificar errores
```

```
            for i in range(len(labels)):
                if predicted[i] != labels[i]:
                    errors.append((images[i].cpu(), metadata[i].cpu(), labels[i].cpu(), predicted[i].cpu()))

    return errors

# Obtener errores
errors = analyze_errors(model, test_loader, device)
print(f"Total de errores: {len(errors)}")
```

4.5. Propuestas de mejoras futuras

Aunque los resultados obtenidos son buenos, identifico áreas de mejora para futuras iteraciones:

- **Experimentar con otras arquitecturas preentrenadas**: Por ejemplo, utilizar EfficientNet o VGG para comparar el rendimiento.
- **Aumentar la cantidad de datos**: Incorporar más datos, si están disponibles, podría mejorar aún más la capacidad de generalización.
- **Implementar técnicas avanzadas de regularización**: Experimentar con L1/L2 regularization o técnicas de early stopping.
- **Explorar explicabilidad del modelo**: Implementar visualizaciones como Grad-CAM para entender cómo el modelo toma decisiones basadas en las imágenes.

4.6. Visualización de features importantes

Para comprender mejor cómo el modelo utiliza los datos estructurados (metadata), generé gráficos de importancia de características basados en la sensibilidad del modelo a las entradas.

Código para analizar importancia de metadata:

```
def analyze_metadata_importance(model, metadata_columns):
    importance = []
    with torch.no_grad():
        for name, param in model.metadata_fc.named_parameters():
            if "weight" in name:
                importance.append(param.abs().mean(dim=0).cpu().numpy())

    importance = np.mean(importance, axis=0)
    plt.bar(metadata_columns, importance)
    plt.title("Importancia de las características de metadata")
    plt.ylabel("Importancia")
    plt.show()

# Visualizar importancia
analyze_metadata_importance(model, metadata_columns)
```