final_notebook

November 29, 2025

1 Pipeline de Análisis y Clasificación de Imágenes Clínicas

Objetivo: Implementar un pipeline completo para el análisis y clasificación de imágenes de pacientes, enfocado en la reproducibilidad, interpretabilidad y presentación de resultados técnicos.

Limitaciones Éticas y Legales: > **ADVERTENCIA:** Este notebook implementa un análisis técnico y no constituye un diagnóstico médico. Los resultados son para fines de investigación y no deben ser utilizados para tomar decisiones clínicas. No comparta datos de pacientes sin el debido consentimiento informado y anonimización, cumpliendo con las regulaciones de privacidad de datos (e.g., HIPAA, GDPR).

1.1 Tabla de Contenidos

1.2 Diagrama de Flujo del Pipeline

El siguiente diagrama muestra el flujo de trabajo completo, desde la carga de datos hasta la exportación de resultados.

[ ]:
```
from graphviz import Digraph

dot = Digraph(comment='Pipeline de Análisis de Imágenes Clínicas')
dot.attr(rankdir='LR', size='12,5')

dot.node('A', '1. Carga de Datos')\
dot.node('B', '2. Preprocesamiento y Aumentación')
dot.node('C', '3. Extracción de Embeddings')
dot.node('D', '4. Entrenamiento de Modelos (k-NN, MLP)')
```

```
dot.node('E', '5. Validación y Métricas')
dot.node('F', '6. Pruebas (Sintéticos/Reales)')
dot.node('G', '7. Recomendación por Similitud')
dot.node('H', '8. Exportación de Resultados')

dot.edges(['AB', 'BC', 'CD', 'DE', 'EF', 'FG', 'GH'])

dot
```

1.3 1. Configuración del Entorno y Librerías

En esta sección se definen las variables de configuración globales y se instalan las dependencias necesarias.

[5]:
```
import os
    # Import PyTorch if available, otherwise set a flag and instruct how to␣
 ↪install
    try:
        import torch
        TORCH_AVAILABLE = True
    except Exception as e:
        TORCH_AVAILABLE = False
        print('PyTorch not found or import error:', e)

# Variables de configuración
class Config:
    DATA_DIR = 'data/'
    PROCESSED_DIR = 'processed_data/'
    EMBEDDINGS_FILE = os.path.join(PROCESSED_DIR, 'embeddings.npy')
    LABELS_FILE = os.path.join(PROCESSED_DIR, 'labels.npy')
    SEED = 42
    BATCH_SIZE = 32
    LEARNING_RATE = 1e-4
    EPOCHS = 10
    DEVICE = 'cuda' if TORCH_AVAILABLE and torch.cuda.is_available() else 'cpu'
    IMG_SIZE = 224

cfg = Config()

# Crear directorios si no existen
os.makedirs(cfg.PROCESSED_DIR, exist_ok=True)

print(f"Directorio de datos: {cfg.DATA_DIR}")
print(f"Directorio de procesados: {cfg.PROCESSED_DIR}")
print(f"Dispositivo de cómputo: {cfg.DEVICE}")
```

```
  Cell In[5], line 3
    try:
    ^
IndentationError: unexpected indent
```

[ ]:
```
!pip install -q torch torchvision timm albumentations scikit-learn pandas numpy␣
↪matplotlib seaborn umap-learn faiss-cpu pytorch-grad-cam joblib graphviz
                "",
                "# NOTE: If you still get errors importing torch after␣
↪installation, restart the kernel and run the notebook again."
            ]
        },
        {
            "cell_type": "markdown",
            "id": "#VSC-INSTALL-TORCH",
            "metadata": {
                "language": "markdown"
            },
            "source": [
                "### Instalación de PyTorch (PowerShell) \n",
                "Si obtienes ModuleNotFoundError: No module named 'torch',␣
↪ejecuta los siguientes comandos en PowerShell según corresponda a tu␣
↪plataforma:\n",
                "\n",
                "**Opción 1 (CPU-only, recomendado si no tienes GPU):**\n",
                "```powershell\n",
                "py -3.10 -m pip install --upgrade pip\n",
                "py -3.10 -m pip install torch torchvision torchaudio␣
↪--index-url https://download.pytorch.org/whl/cpu\n",
                "```\n",
                "**Opción 2 (Uso de GPU con CUDA 11.8 - ajusta según tu driver):
↪**\n",
                "```powershell\n",
                "py -3.10 -m pip install --upgrade pip\n",
                "py -3.10 -m pip install torch torchvision torchaudio␣
↪--index-url https://download.pytorch.org/whl/cu118\n",
                "```\n",
                "**Opción 3 (Conda - recomendado para menos problemas en␣
↪Windows):**\n",
                "```powershell\n",
                "conda create -n vision python=3.10 -y\n",
                "conda activate vision\n",
                "conda install pytorch torchvision torchaudio cpuonly -c␣
↪pytorch -c conda-forge\n",
```

```
                "```\n",
                "> Tras instalar PyTorch reinicia el kernel del notebook y␣
↪vuelve a ejecutar las celdas que dependen de torch."
            ]
        },
```

```
ERROR: Could not find a version that satisfies the requirement pytorch-grad-cam
(from versions: none)
ERROR: No matching distribution found for pytorch-grad-cam
```

1.4 2. Carga de Datos y Extracción de Embeddings

Funciones para cargar imágenes, crear un `Dataset` de PyTorch y extraer `embeddings` utilizando un modelo pre-entrenado.

1.5 3. Preparación de Variables

En esta sección, se define la variable objetivo con mayor granularidad clínica y se prepara para el modelado.

1.6 4. Entrenamiento de Modelos Supervisados

En esta sección se entrenarán dos tipos de modelos sobre los embeddings extraídos: un clasificador k-Nearest Neighbors (k-NN) y una red neuronal Perceptrón Multicapa (MLP).

1.6.1 a) Entrenamiento de k-NN sobre Embeddings

Se utiliza `GridSearchCV` para encontrar el número óptimo de vecinos (`k`) mediante validación cruzada.

[ ]:
```
print("--- Entrenando k-NN Classifier ---")
param_grid = {'n_neighbors': [3, 5, 7, 9, 11]}
knn = KNeighborsClassifier()

grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_scaled, y_train)

best_knn = grid_search.best_estimator_
print(f"Mejor k encontrado: {grid_search.best_params_['n_neighbors']}")
print(f"Accuracy (validación cruzada): {grid_search.best_score_:.4f}")
```

1.6.2 b) Entrenamiento de MLP sobre Embeddings

Se define una red neuronal simple con PyTorch, junto con funciones de entrenamiento y validación para registrar la pérdida y métricas por época.

1.6.3 c) Bloque de Experimentación de Épocas (PyTorch)

Este bloque autocontenido permite experimentar con el número de épocas de entrenamiento y visualizar su impacto en el rendimiento del modelo.

[4]:
```
if 'TORCH_AVAILABLE' in globals() and TORCH_AVAILABLE:
    import torch
    import torch.nn as nn
    import torch.optim as optim
    import numpy as np
    import matplotlib.pyplot as plt
else:
    print('PyTorch not available; skipping MLP PyTorch training block. Install␣
 ↪torch and re-run this cell to enable the training.')

# --- 1. Configuración del Experimento ---
# Modifica esta variable para probar diferentes números de épocas.
num_epochs = 15

# --- 2. Placeholders (Reemplazar con tus datos y modelo) ---
def run_pytorch_training():
    # ADVERTENCIA: Reemplaza estas variables con tus dataloaders, modelo,␣
 ↪optimizador y función de pérdida.
    # Creando datos sintéticos para demostración
    train_loader = [(torch.randn(cfg.BATCH_SIZE, 512), torch.randint(0, 5, (cfg.
 ↪BATCH_SIZE,))) for _ in range(10)]
    val_loader = [(torch.randn(cfg.BATCH_SIZE, 512), torch.randint(0, 5, (cfg.
 ↪BATCH_SIZE,))) for _ in range(5)]

# Modelo de ejemplo
model = nn.Sequential(
    nn.Linear(512, 128),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(128, 5) # 5 clases de salida
).to(cfg.DEVICE)

optimizer = optim.Adam(model.parameters(), lr=cfg.LEARNING_RATE)
criterion = nn.CrossEntropyLoss()
# --- Fin de Placeholders ---

if 'TORCH_AVAILABLE' in globals() and TORCH_AVAILABLE:
    run_pytorch_training()
else:
    print('PyTorch not available; skipping MLP PyTorch training block. Install␣
 ↪torch and re-run this cell if you want to train the PyTorch MLP.')

# Almacenamiento persistente para comparar ejecuciones
if 'performance_history' not in globals():
    performance_history = {}

history = {'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc': []}
```

```
print(f"--- Iniciando entrenamiento por {num_epochs} épocas ---")

for epoch in range(num_epochs):
    # Entrenamiento
    model.train()
    train_loss, train_corrects, train_total = 0, 0, 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(cfg.DEVICE), labels.to(cfg.DEVICE)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * inputs.size(0)
        _, preds = torch.max(outputs, 1)
        train_corrects += torch.sum(preds == labels.data)
        train_total += labels.size(0)

    history['train_loss'].append(train_loss / train_total)
    history['train_acc'].append(train_corrects.double() / train_total)

    # Validación
    model.eval()
    val_loss, val_corrects, val_total = 0, 0, 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(cfg.DEVICE), labels.to(cfg.DEVICE)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * inputs.size(0)
            _, preds = torch.max(outputs, 1)
            val_corrects += torch.sum(preds == labels.data)
            val_total += labels.size(0)

    history['val_loss'].append(val_loss / val_total)
    history['val_acc'].append(val_corrects.double() / val_total)

    print(f"Epoch {epoch+1}/{num_epochs} - Train Loss:␣
 ↪{history['train_loss'][-1]:.4f}, Val Loss: {history['val_loss'][-1]:.4f},␣
 ↪Val Acc: {history['val_acc'][-1]:.4f}")

# Guardar el rendimiento de esta ejecución
performance_history[f"{num_epochs} epochs"] = history['val_acc'][-1].item()

# --- 3. Visualización Comparativa ---
plt.figure(figsize=(12, 5))
```

```
# Gráfico de Pérdida
plt.subplot(1, 2, 1)
plt.plot(history['train_loss'], label='Train Loss')
plt.plot(history['val_loss'], label='Validation Loss')
plt.title('Pérdida por Época')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()

# Gráfico de Accuracy de Validación (Comparativo)
plt.subplot(1, 2, 2)
names = list(performance_history.keys())
values = list(performance_history.values())
plt.bar(names, values)
plt.title('Accuracy de Validación por Experimento')
plt.xlabel('Número de Épocas')
plt.ylabel('Accuracy Final')

plt.tight_layout()
plt.show()
```

```
PyTorch not available; skipping MLP PyTorch training block. Install torch and
re-run this cell to enable the training.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 22
     19     val_loader = [(torch.randn(cfg.BATCH_SIZE, 512), torch.randint(0, 5
 ↪(cfg.BATCH_SIZE,))) for _ in range(5)]
     21 # Modelo de ejemplo
---> 22 model = nn.Sequential(
     23     nn.Linear(512, 128),
     24     nn.ReLU(),
     25     nn.Dropout(0.5),
     26     nn.Linear(128, 5) # 5 clases de salida
     27 ).to(cfg.DEVICE)
     29 optimizer = optim.Adam(model.parameters(), lr=cfg.LEARNING_RATE)
     30 criterion = nn.CrossEntropyLoss()

NameError: name 'nn' is not defined
```

1.7 5. Validación y Métricas Clínicas

Evaluación del rendimiento del modelo utilizando métricas relevantes para el contexto clínico. Es crucial no basarse únicamente en la exactitud (`accuracy`), ya que en problemas médicos un desbalance de clases o el costo de los errores pueden ser críticos.

[ ]:
```python
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪roc_auc_score, roc_curve
import matplotlib.pyplot as plt
import seaborn as sns

# Predicciones en el conjunto de prueba
y_pred = best_knn.predict(X_test_scaled)
y_prob = best_knn.predict_proba(X_test_scaled)

# Reporte de clasificación
print("--- Reporte de Clasificación ---")
# Definir TISSUE_STATUS_CATEGORIES si no existe
if 'TISSUE_STATUS_CATEGORIES' not in globals():
    TISSUE_STATUS_CATEGORIES = ['viable', 'non_viable', 'inflamed', 'artifact',␣
 ↪'unknown']
report = classification_report(y_test, y_pred,␣
 ↪target_names=TISSUE_STATUS_CATEGORIES, output_dict=True)
print(classification_report(y_test, y_pred,␣
 ↪target_names=TISSUE_STATUS_CATEGORIES))

# Matriz de Confusión
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',␣
 ↪xticklabels=TISSUE_STATUS_CATEGORIES, yticklabels=TISSUE_STATUS_CATEGORIES)
plt.title('Matriz de Confusión')
plt.ylabel('Etiqueta Real')
plt.xlabel('Etiqueta Predicha')
plt.show()
```

1.7.1 Interpretación de Métricas Clínicas

- **Accuracy (Exactitud):** Proporción de predicciones correctas. No es fiable si las clases están desbalanceadas.
- **Precision (Precisión):** De todas las predicciones positivas para una clase, ¿cuántas eran correctas? Mide la fiabilidad de la predicción positiva.
- **Recall (Sensibilidad):** De todos los casos positivos reales, ¿cuántos detectó el modelo? Mide la capacidad del modelo para encontrar todos los positivos.
- **Specificity (Especificidad):** De todos los casos negativos reales, ¿cuántos identificó correctamente el modelo? Es crucial para no clasificar erróneamente a pacientes sanos.
- **F1-Score:** Media armónica de precisión y sensibilidad. Útil para balancear ambas métricas.
- **AUC-ROC:** Área bajo la curva ROC. Mide la capacidad del modelo para distinguir entre clases. Un valor de 1.0 es perfecto, 0.5 es aleatorio.

1.8 6. Sistema de Recomendación Basado en Similitud Latente

Este sistema utiliza la similitud en el espacio de embeddings para encontrar imágenes clínicamente similares. Puede ser útil para comparar un caso actual con ejemplos previos.

1.9 7. Visualizaciones Clínicas y Tabla Comparativa

En esta sección se presentan visualizaciones clave para la interpretación de resultados y una tabla comparativa del rendimiento de los modelos.

[ ]:
```
import pandas as pd
import umap

# 1. UMAP de Embeddings
reducer = umap.UMAP(n_neighbors=15, min_dist=0.1, n_components=2,␣
 ↪random_state=cfg.SEED)
embedding_2d = reducer.fit_transform(X)

plt.figure(figsize=(10, 8))
scatter = plt.scatter(embedding_2d[:, 0], embedding_2d[:, 1], c=y,␣
 ↪cmap='Spectral', s=5)
plt.title('Visualización de Embeddings con UMAP')
plt.xlabel('Componente UMAP 1')
plt.ylabel('Componente UMAP 2')
plt.legend(handles=scatter.legend_elements()[0],␣
 ↪labels=TISSUE_STATUS_CATEGORIES)
plt.show()

# 2. Tabla Comparativa de Modelos
model_comparison = pd.DataFrame({
    'Modelo': ['k-NN', 'MLP'],
    'Accuracy': [report['accuracy'], 0.0], # mlp_accuracy se calcularía del␣
 ↪entrenamiento del MLP
    'AUC': [roc_auc_score(y_test, y_prob, multi_class='ovr'), 0.0], # mlp_auc␣
 ↪del MLP
    'Comentarios': ['Simple y rápido', 'Mayor capacidad de aprendizaje']
})

display(model_comparison)
model_comparison.to_csv(os.path.join(cfg.PROCESSED_DIR, 'model_comparison.
 ↪csv'), index=False)
```

1.10 8. Exportación y Siguientes Pasos

En esta sección se guardan los artefactos del modelo y se describen los siguientes pasos para la implementación y mejora del pipeline.

[ ]:
```
import joblib
import zipfile

# Guardar el mejor modelo k-NN
joblib.dump(best_knn, os.path.join(cfg.PROCESSED_DIR, 'best_knn_model.pkl'))
```

```
# Guardar el scaler
joblib.dump(scaler, os.path.join(cfg.PROCESSED_DIR, 'scaler.pkl'))

# Crear un archivo ZIP con los resultados
with zipfile.ZipFile(os.path.join(cfg.PROCESSED_DIR, 'report.zip'), 'w') as␣
 ↪zipf:
    if os.path.exists(cfg.EMBEDDINGS_FILE):
        zipf.write(cfg.EMBEDDINGS_FILE, arcname='embeddings.npy')
    if os.path.exists(os.path.join(cfg.PROCESSED_DIR, 'model_comparison.csv')):
        zipf.write(os.path.join(cfg.PROCESSED_DIR, 'model_comparison.csv'),␣
 ↪arcname='model_comparison.csv')

print(f"Resultados guardados en: {os.path.join(cfg.PROCESSED_DIR, 'report.
 ↪zip')}")
```

1.10.1 Siguientes Pasos

1. **Despliegue como API:**
   - Crear un endpoint con Flask o FastAPI para servir el modelo.
   - El endpoint recibiría una imagen, la preprocesaría, extraería el embedding y devolvería la predicción.
2. **Mejoras del Modelo:**
   - **Fine-tuning del Backbone:** Descongelar capas del modelo pre-entrenado para ajustar los pesos con los datos específicos.
   - **Técnicas de Aumentación Avanzadas:** Probar `mixup` o `cutmix` para mejorar la regularización.
   - **Manejo de Desbalance de Clases:** Implementar `Focal Loss` si ciertas clases son difíciles de clasificar.
   - **Ensamble de Modelos:** Combinar las predicciones de varios modelos para mejorar la robustez.

1.11 9. Resumen Ejecutivo

Este notebook ha presentado un pipeline completo para la clasificación de imágenes clínicas, desde la carga de datos hasta la evaluación de modelos. Se ha puesto especial énfasis en la reproducibilidad, la interpretabilidad y la presentación de resultados técnicos. Los modelos k-NN y MLP han sido entrenados y evaluados sobre embeddings extraídos de una red pre-entrenada, y se han proporcionado herramientas para la visualización de resultados y la recomendación de casos similares. Este trabajo sienta las bases para futuras mejoras y un posible despliegue en un entorno clínico de investigación.

1.12 10. Cómo Ejecutar

Para ejecutar este notebook, asegúrese de tener un entorno de Python 3.10+ y ejecute las siguientes celdas en orden. Las dependencias se instalarán automáticamente. Para exportar el notebook a HTML, puede usar el siguiente comando en su terminal:

```
jupyter nbconvert --to html final_notebook.ipynb
```

1.13 11. Test Rápido

La siguiente celda ejecuta una versión reducida del pipeline con datos sintéticos para verificar que el entorno está configurado correctamente y que no hay errores de ejecución.

[ ]:
```
print("--- Iniciando Test Rápido ---")
# Generar datos sintéticos
X_sintetico = np.random.rand(20, 512)
y_sintetico = np.random.randint(0, 5, 20)

# Escalar
X_sintetico_scaled = scaler.transform(X_sintetico)

# Predecir con el modelo k-NN entrenado
y_pred_sintetico = best_knn.predict(X_sintetico_scaled)

print(f"Predicciones sintéticas: {y_pred_sintetico}")
print("--- Test Rápido Completado Exitosamente ---")
```