Documentación Técnica de la Estructura del Proyecto

Este documento describe la organización del proyecto y detalla las funciones principales de cada archivo. Además, se incluyen **consideraciones de mejora** para futuras iteraciones, como la integración de MLflow y la posibilidad de realizar *fine-tuning* sobre un *frontier model* (por ejemplo, "gpt-4o-mini-2024-07-18").

---

1. Estructura General del Proyecto

```
bash
Copy
ml_test/                  # Entorno virtual (virtualenv) con dependencias
instaladas
data/
│   └── apli_challenge_data.csv         # Datos originales sin procesar
notebooks/
│   └── 1_EDA.ipynb                     # Análisis exploratorio de datos
y visualizaciones iniciales
src/
│   ├── __init__.py
│   ├── data_preprocessing.py           # Clase para limpieza y
transformación de textos
│   ├── model.py                        # Clase(s) y funciones
relacionadas con el modelo
│   ├── evaluation.py                   # Clase(s) para la evaluación
(ROUGE, etc.)
│   └── pipeline/
│       └── pipeline.py                 # Pipeline principal que integra
las etapas y se ejecuta desde main.py
results/
│   ├── testing_pipeline_v1.ipynb       # Notebook ejecutado en Google
Colab - modelo 1
│   ├── testing_pipeline_v2.ipynb       # Notebook ejecutado en Google
Colab - modelo 2
│   └── testing_pipeline_v3.ipynb       # Notebook ejecutado en Google
Colab - modelo 2 con 2 épocas de entrenamiento
main.py                                 # Punto de entrada para ejecutar
el pipeline
```

Breve Descripción de Carpetas

- **data**: Contiene los datos en bruto (CSV original).
- **notebooks**: Notebooks de análisis y visualización.
- **src**: Código fuente organizado en módulos (preprocesamiento, modelo, evaluación, pipeline).

- **results**: Notebooks donde se han hecho pruebas de la *pipeline* en Google Colab, guardando resultados y experimentos.
- **main.py**: Archivo principal que orquesta la ejecución del pipeline (preprocesamiento, entrenamiento, evaluación).

---

2. Contenido de Cada Archivo

2.1 `data_preprocessing.py`

- **Responsabilidad**:
  - Realizar limpieza de texto (conversión a minúsculas, eliminación de HTML, caracteres especiales).
  - Eliminar duplicados y, en general, preparar el DataFrame antes de pasarlo al modelo.
- **Clases/Funciones Clave**:
  - `DataPreprocessor`: Contiene métodos como `load_data()`, `apply_cleaning()`, `remove_duplicates()` y `get_cleaned_data()`.

2.2 `model.py`

- **Responsabilidad**:
  - Definir clases o funciones para la configuración del modelo de *summarization*.
  - Cargar modelos preentrenados de Hugging Face (por ejemplo, T5, BART) y configurar *fine-tuning* básico.
- **Clases/Funciones Clave**:
  - `TransformerFineTuner`: Gestiona la creación del tokenizer, el modelo base, la preparación del dataset y la configuración del *Trainer* de Hugging Face.

2.3 `evaluation.py`

- **Responsabilidad**:
  - Generar predicciones sobre el conjunto de validación o test.
  - Calcular métricas (p. ej., ROUGE) y analizar la distribución de los puntajes.
- **Clases/Funciones Clave**:
  - `ModelEvaluator`: Contiene métodos como `generate_predictions()` y `evaluate()`, además de utilidades para graficar la distribución de ROUGE y mostrar ejemplos con bajo puntaje.

2.4 `pipeline/pipeline.py`

- **Responsabilidad**:

    - Integrar las distintas etapas (preprocesamiento, entrenamiento, evaluación) en un solo flujo.
- **Clases/Funciones Clave**:
    - `FullPipeline`: Ejecuta los pasos en orden (carga y limpieza de datos, setup del modelo, fine-tuning, evaluación, etc.) dentro de un método como `run_pipeline()`.

2.5 `main.py`

- **Responsabilidad**:
    - Punto de entrada de la aplicación, donde se instancia la clase `FullPipeline` y se llama a `run_pipeline()` para ejecutar todo el proceso de principio a fin.
- **Uso**:
    - Desde la línea de comandos: `python main.py`

2.6 `1_EDA.ipynb` (en `notebooks/`)

- **Responsabilidad**:
    - Exploración inicial de datos (visualizaciones, estadísticas descriptivas, histogramas, word clouds).
    - Sirve para comprender la distribución y características del dataset antes de entrenar el modelo.

2.7 `results/` (Carpeta de Notebooks de Prueba)

- `testing_pipeline_v1.ipynb`, `testing_pipeline_v2.ipynb`, `testing_pipeline_v3.ipynb`:
    - Contienen experimentos y pruebas ejecutadas en Google Colab, registrando resultados de distintas configuraciones (diferentes modelos, épocas de entrenamiento, etc.).

---

3. Consideraciones y Posibles Mejoras

1. **Integración de MLflow**
    - Permitiría **trackear experimentos** de forma sistemática:
        - Registrar automáticamente parámetros (número de épocas, *learning rate*, etc.), métricas (ROUGE, pérdida), artefactos (pesos del modelo) y versiones del dataset.
    - Beneficia la **reproducibilidad** y la comparación de múltiples experimentos.
2. **Fine-Tuning a un "Frontier Model"** (p. ej., `gpt-4o-mini-2024-07-18`)
    - Modelos más ligeros y optimizados podrían reducir tiempos de entrenamiento e inferencia.

    - Requiere ajustar la clase `TransformerFineTuner` para cargar el nuevo modelo y, posiblemente, adecuar la tokenización o configuración del *Trainer* si el modelo tiene requisitos especiales.
3. **Uso de un Archivo de Configuración**
    - Centralizar parámetros (rutas de datos, hiperparámetros, nombre del modelo, etc.) en un archivo YAML o JSON.
    - Evitar *hardcoding* de valores en el código y facilitar la ejecución en distintos entornos (local, Colab, etc.).
4. **Optimización de la Decodificación**
    - Ajustar parámetros como `num_beams`, `temperature` o `top_k` para encontrar el balance entre precisión y diversidad en los resúmenes.
5. **Automatización de la Limpieza y Filtrado de Outliers**
    - Añadir pasos opcionales para truncar textos demasiado largos o descartar artículos con menos de X palabras.
    - Mejora la consistencia del dataset y puede reducir el ruido durante el entrenamiento.

---

4. Conclusión

La estructura de este proyecto sigue un **patrón modular**, separando claramente la etapa de preprocesamiento, la lógica del modelo y la evaluación en diferentes archivos. Esto facilita el mantenimiento y la extensibilidad del código.

Para escalar o mejorar el sistema, se sugiere:

- **Integrar MLflow** para un mejor control de experimentos.
- **Probar un frontier model** (p. ej. `gpt-4o-mini-2024-07-18`) si se busca más eficiencia o mejores resultados en el *fine-tuning*.
- **Centralizar configuraciones** y parámetros para facilitar la reproducción y despliegue del pipeline.