Okay, Jim, let's take a fresh, structured approach to this complex task. This is a very common and valuable problem in scientific text analysis. You're right, the nuances of identifying and classifying datasets require more than just simple regexes. An LLM is definitely the way to go for the classification part.

Here's a comprehensive outline for your Python notebook, designed to run both locally and in an offline Kaggle environment, followed by detailed explanations and code snippets for each section.

Project Outline: Academic Article Dataset Analyzer

Goal: Train an open-source LLM to identify and classify datasets within academic research articles (PDF/XML).

Output: A structured list (e.g., CSV) of `article_id`, `dataset_id`, and `classification_label` ("Primary", "Secondary", "Missing").

Phase 0: Environment Setup & Offline Preparation

1. **Kaggle Dataset Strategy:** How to get all necessary files (articles, training data, LLM, libraries) into the offline Kaggle environment.
2. **Required Libraries:** List of Python packages.

Phase 1: Data Loading & Preprocessing

1. **Load Article Content:**
    - Function to extract text from PDF files.
    - Function to extract text from XML files.
    - Store extracted text in a dictionary or DataFrame, keyed by `article_id`.
2. **Load Labeled Training Data:**
    - Read your `article_id, dataset_id, label` CSV.

Phase 2: Information Extraction (IE) - Dataset Identification

1. **Strategy:** Combine rule-based (regex) and potentially LLM-assisted (if needed) methods to find dataset identifiers.
2. **Implementation:**
    - Function to extract common dataset identifiers (DOIs, accession numbers, specific repository URLs) using robust regex patterns.
    - Handle cases where an article might have zero, one, or multiple dataset IDs.

Phase 3: LLM Model Training (Fine-tuning)

1. **Data Preparation for Fine-tuning:**
    - Transform your labeled CSV data into the ChatML format expected by Qwen, incorporating article text snippets.
    - Create a Hugging Face `Dataset` object.
2. **Load Base Qwen Model:**
    - Load the pre-downloaded Qwen model and tokenizer from your Kaggle Dataset.
3. **Configure & Run Supervised Fine-Tuning (SFT):**
    - Use `trl.SFTTrainer` for efficient fine-tuning.
    - Define `TrainingArguments` suitable for Kaggle's resources.
4. **Save Fine-tuned Model:**
    - Save the trained model and tokenizer to `/kaggle/working/` for later use.

Phase 4: LLM-based Classification (Inference)

1. **Load Fine-tuned Model:**
    - Load your saved fine-tuned Qwen model and tokenizer.

2. **Prompt Engineering for Classification:**
    - Craft effective prompts for the LLM to classify each identified `dataset_id` as "Primary" or "Secondary", providing relevant article context.
3. **Inference Loop:**
    - Iterate through each article.
    - For each article, iterate through its identified dataset IDs.
    - Call the LLM for classification.
4. **Handle "Missing" Classification:**
    - If Phase 2 (IE) finds no dataset IDs for an article, directly label it as "Missing".

Phase 5: Results & Output

1. **Consolidate Results:**
    - Store all `article_id`, `dataset_id`, `classification_label` entries in a Pandas DataFrame.
2. **Save Output:**
    - Save the final DataFrame to a CSV file in `/kaggle/working/`.

Detailed Implementation & Code Snippets

Phase 0: Environment Setup & Offline Preparation

Kaggle Dataset Strategy:
For an offline Kaggle notebook, everything must be available as a Kaggle Dataset.
1. **Qwen Model:** Download the specific Qwen model (e.g., `Qwen/Qwen1.5-1.8B-Chat`) from Hugging Face. Create a Kaggle Dataset from these downloaded files.
2. **Libraries:** If `PyMuPDF`, `trl`, `accelerate`, `bitsandbytes` (for 8-bit

optimization) are not pre-installed or you need specific versions, download their `.whl` files (e.g., from PyPI) and create Kaggle Datasets. You'll install them with `!pip install /kaggle/input/your-lib-whl-dataset/*.whl --no-index --find-links /kaggle/input/your-lib-whl-dataset/`.
3. **Your Articles:** Upload your PDF and XML article files as a Kaggle Dataset.
4. **Your Training Data:** Upload your `training_data.csv` as a Kaggle Dataset.

Notebook Setup (Initial Cell):

```
# --- 0. Environment Setup & Offline Preparation ---
# This cell primarily defines paths and ensures libraries are available.

# Define paths to your Kaggle Datasets
# IMPORTANT: Adjust these paths to match where your datasets are mounted
in Kaggle
QWEN_BASE_MODEL_PATH = "/kaggle/input/qwen1-5-1-8b-chat-hf" # Example
path for Qwen model
ARTICLES_DIR = "/kaggle/input/your-articles-dataset/articles/" # Path to
your PDF/XML articles
TRAINING_DATA_CSV_PATH = "/kaggle/input/your-training-data/
training_data.csv" # Path to your labeled CSV

# Output directory for the fine-tuned model and results
FINE_TUNED_MODEL_OUTPUT_DIR = "/kaggle/working/
qwen_finetuned_dataset_classifier"
FINAL_RESULTS_CSV_PATH = "/kaggle/working/
article_dataset_classification.csv"

# Install necessary libraries from Kaggle Datasets if not pre-installed
# Example for PyMuPDF (fitz) - you'd need to upload its wheel file
# !pip install /kaggle/input/pymupdf-whl/*.whl --no-index --find-links /
kaggle/input/pymupdf-whl/

# For bitsandbytes (optional, for 8-bit quantization during training)
# !pip install /kaggle/input/bitsandbytes-0-41-1-py3-none-any-whl/*.whl
--no-index --find-links /kaggle/input/bitsandbytes-0-41-1-py3-none-any-
whl/

# For trl (Supervised Fine-Tuning)
# !pip install /kaggle/input/trl-0-7-10-py3-none-any-whl/*.whl --no-index
--find-links /kaggle/input/trl-0-7-10-py3-none-any-whl/

# Standard Imports
```

```
import os
import glob
import re
import pandas as pd
import xml.etree.ElementTree as ET
import collections # For deque in parenthesis removal

# Conditional imports for external libraries
try:
    import fitz # PyMuPDF for PDF processing
except ImportError:
    print("PyMuPDF (fitz) not found. PDF processing will be skipped.")
    fitz = None

try:
    from transformers import AutoModelForCausalLM, AutoTokenizer, Trainin
gArguments
    from trl import SFTTrainer
    import torch
    # Optional: for 8-bit training
    # import bitsandbytes as bnb
except ImportError:
    print("Transformers, TRL, or PyTorch not found. LLM training/
inference will be skipped.")
    AutoModelForCausalLM, AutoTokenizer, TrainingArguments, SFTTrainer, t
orch = None, None, None, None, None
    # bnb = None

# Set device for PyTorch
device = "cuda" if torch and torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")
```

Phase 1: Data Loading & Preprocessing

```
# --- 1. Data Loading & Preprocessing ---

def read_pdf_text(pdf_path: str) -> str | None:
    """Extracts all text from a PDF file using PyMuPDF."""
    if not fitz:
        return None
    try:
        doc = fitz.open(pdf_path)
        text = ""
```

```
        for page in doc:
            text += page.get_text()
        doc.close()
        return text
    except Exception as e:
        print(f"Error reading PDF {pdf_path}: {e}")
        return None

def read_xml_text(xml_file_path: str) -> str | None:
    """Reads and concatenates all text content from an XML file."""
    all_text_parts = []
    try:
        tree = ET.parse(xml_file_path)
        root = tree.getroot()
        for element in root.iter():
            if element.text:
                cleaned_text = element.text.strip()
                if cleaned_text:
                    all_text_parts.append(cleaned_text)
            if element.tail:
                cleaned_tail = element.tail.strip()
                if cleaned_tail:
                    all_text_parts.append(cleaned_tail)
        return " ".join(all_text_parts) if all_text_parts else ""
    except Exception as e:
        print(f"Error reading XML {xml_file_path}: {e}")
        return None

def load_articles(articles_dir: str) -> dict[str, str]:
    """
    Loads text content from all PDF and XML files in the specified
directory.
    Returns a dictionary mapping article_id (filename without extension)
to its text content.
    """
    article_texts = {}
    article_files = glob.glob(os.path.join(articles_dir, "*.pdf")) + \
                    glob.glob(os.path.join(articles_dir, "*.xml"))

    print(f"Found {len(article_files)} article files.")

    for filepath in article_files:
        article_id = os.path.splitext(os.path.basename(filepath))[0]
        text_content = None
```

```
        if filepath.endswith(".pdf"):
            text_content = read_pdf_text(filepath)
        elif filepath.endswith(".xml"):
            text_content = read_xml_text(filepath)

        if text_content:
            article_texts[article_id] = text_content
        else:
            print(f"Warning: Could not extract text from {filepath}.
Skipping.")

    print(f"Successfully loaded text for {len(article_texts)} articles.")
    return article_texts

# Load all article texts once
all_article_texts = load_articles(ARTICLES_DIR)

# Load labeled training data
try:
    training_df = pd.read_csv(TRAINING_DATA_CSV_PATH)
    print(f"Loaded {len(training_df)} labeled training examples.")
    print("Training data head:")
    print(training_df.head())
except FileNotFoundError:
    print(f"Training data CSV not found at {TRAINING_DATA_CSV_PATH}.
Skipping training phase.")
    training_df = pd.DataFrame() # Empty DataFrame if not found
```

Phase 2: Information Extraction (IE) - Dataset Identification

This is where the "zero or more datasets" comes in. We'll use a set of robust regexes.

```
# --- 2. Information Extraction (IE) - Dataset Identification ---

# Regex patterns for common dataset identifiers
# This list can be expanded based on the types of IDs you expect.
# Using raw strings and including common variations for hyphens/dashes.
DOI_PATTERN = r'\b10\.\d{4,9}/[-._;()/:A-Za-
z0-9\u002D\u2010\u2011\u2012\u2013\u2014\u2015]+\b'
# Example for GenBank accession numbers (e.g., AB123456, AF000001)
GENBANK_PATTERN = r'\b[A-Z]{1,2}\d{5,6}\b'
# Example for GEO accession numbers (e.g., GSE12345, GSM12345)
```

```
GEO_PATTERN = r'\b(GSE|GSM|GDS|GPL|GSE|GPL)\d{4,6}\b'
# Example for Dryad DOIs (often follow a specific pattern)
DRYAD_DOI_PATTERN = r'\b10\.5061/dryad\.[a-zA-Z0-9]{5,}\b' # e.g.,
10.5061/dryad.2bs69

# Combine all patterns into a list
DATASET_ID_PATTERNS = [
    DOI_PATTERN,
    GENBANK_PATTERN,
    GEO_PATTERN,
    DRYAD_DOI_PATTERN,
    # Add more as needed, e.g., for specific repositories like Figshare,
Zenodo, etc.
]

# Compile all patterns for efficiency
COMPILED_DATASET_ID_REGEXES = [re.compile(p) for p in
DATASET_ID_PATTERNS]

def extract_dataset_ids_from_text(text: str) -> list[str]:
    """
    Extracts potential dataset identifiers from the given text using a
list of regex patterns.
    """
    found_ids = set() # Use a set to store unique IDs
    if not text:
        return []

    # Remove non-matching parentheses first, as they can interfere with
DOI matching
    # (This uses the function you asked for previously)
    def remove_unmatched_parentheses_local(s: str) -> str:
        open_paren_indices_stack = collections.deque()
        keep_char = [True] * len(s)
        for i, char in enumerate(s):
            if char == '(':
                open_paren_indices_stack.append(i)
            elif char == ')':
                if open_paren_indices_stack:
                    open_paren_indices_stack.pop()
                else:
                    keep_char[i] = False
        while open_paren_indices_stack:
            unmatched_open_idx = open_paren_indices_stack.pop()
            keep_char[unmatched_open_idx] = False
```

```
        return "".join([s[i] for i, should_keep in enumerate(keep_char) i
f should_keep])

    cleaned_text = remove_unmatched_parentheses_local(text)

    for compiled_regex in COMPILED_DATASET_ID_REGEXES:
        for match in compiled_regex.finditer(cleaned_text):
            found_ids.add(match.group(0))

    return list(found_ids)

# Example of how to use it (will be integrated into main processing loop
later)
# article_id_example = list(all_article_texts.keys())[0]
# text_example = all_article_texts[article_id_example]
# extracted_ids = extract_dataset_ids_from_text(text_example)
# print(f"\nExtracted IDs from '{article_id_example}': {extracted_ids}")
```

Phase 3: LLM Model Training (Fine-tuning)

This phase will only run if `training_df` is not empty and LLM components are available.

```
# --- 3. LLM Model Training (Fine-tuning) ---

# Global variables for LLM components
llm_tokenizer = None
llm_model = None

def load_base_llm_for_training():
    global llm_tokenizer, llm_model
    if not AutoModelForCausalLM or not QWEN_BASE_MODEL_PATH:
        print("LLM components not available or base model path not set.
Skipping LLM loading.")
        return False
    try:
        print(f"Loading Qwen tokenizer from: {QWEN_BASE_MODEL_PATH}")
        llm_tokenizer = AutoTokenizer.from_pretrained(QWEN_BASE_MODEL_PAT
H, trust_remote_code=True)
        if llm_tokenizer.pad_token is None:
            llm_tokenizer.pad_token = llm_tokenizer.eos_token
            print("Set tokenizer.pad_token to tokenizer.eos_token")
```

```
        print(f"Loading Qwen model from: {QWEN_BASE_MODEL_PATH}")
        llm_model = AutoModelForCausalLM.from_pretrained(
            QWEN_BASE_MODEL_PATH,
            torch_dtype=torch.bfloat16 if torch.cuda.is_available() and t
orch.cuda.is_bf16_supported() else torch.float32,
            device_map="auto", # Automatically uses GPU if available
            trust_remote_code=True,
            # load_in_8bit=True if bnb else False # Uncomment if
bitsandbytes is used
        )
        print(f"Base LLM loaded successfully on {llm_model.device}.")
        return True
    except Exception as e:
        print(f"Error loading base LLM for training: {e}")
        llm_tokenizer, llm_model = None, None # Reset to None on failure
        return False

def create_finetuning_prompt_chatml(article_snippet: str, dataset_id:
str, label: str) -> str:
    """
    Creates a prompt in Qwen's ChatML format for fine-tuning.
    The SFTTrainer will handle masking the prompt part for loss
calculation.
    """
    user_message = f"""
Article Context (excerpt):
"{article_snippet}"

Dataset Identifier: "{dataset_id}"

Question: Based on the provided article context, was the dataset
(identified as "{dataset_id}"):
1. Created by the authors primarily for the research described in THIS
article? (If so, it's "Primary")
2. An existing dataset that the authors obtained and used for their
research in THIS article? (If so, it's "Secondary")

Please respond with only one word: "Primary" or "Secondary".
"""
    # ChatML format: system message, user message, assistant response
    return f"<|im_start|>system\nYou are an expert research assistant.<|
im_end|>\n<|im_start|>user\n{user_message.strip()}<|im_end|>\n<|im_start|
>assistant\n{label}<|im_end|>"

if not training_df.empty and llm_model is None:
```

```
# Only attempt to load if training data exists and model not loaded
    load_base_llm_for_training()

if llm_model and not training_df.empty:
    print("\n--- Preparing data for Fine-tuning ---")
    formatted_texts = []
    for _, row in training_df.iterrows():
        article_id = row['article_id']
        dataset_id = row['dataset_id']
        label = row['label'] # "Primary" or "Secondary"

        # Get the full article text. Truncate if too long for context
window.
        # Qwen 1.5 models typically have 32k context, but for training,
shorter is faster.
        # Adjust max_length based on your model and GPU memory.
        article_text = all_article_texts.get(article_id, "")
        if not article_text:
            print(f"Warning: Article text for {article_id} not found.
Skipping training example.")
            continue

        # Truncate article text to fit within context window for training
        # A common practice is to take a snippet around the dataset
mention if possible,
        # but for simplicity here, we'll just take the beginning.
        max_context_length = 2048 # Adjust based on model and VRAM
        truncated_article_text = article_text[:max_context_length]

        formatted_texts.append({"text": create_finetuning_prompt_chatml(t
runcated_article_text, dataset_id, label)})

    if formatted_texts:
        from datasets import Dataset
        train_dataset = Dataset.from_list(formatted_texts)
        print(f"Prepared {len(train_dataset)} examples for fine-tuning.")
        print("Example formatted training instance:")
        print(train_dataset[0]['text'])

        print("\n--- Starting Fine-tuning ---")
        try:
            training_args = TrainingArguments(
                output_dir=f"{FINE_TUNED_MODEL_OUTPUT_DIR}/checkpoints",
                num_train_epochs=1,  # Start with 1 epoch, adjust as
needed
```

```
                per_device_train_batch_size=1, # Adjust based on VRAM
                gradient_accumulation_steps=4,
# Effective batch size = 1 * 4 = 4
                learning_rate=2e-5,
                logging_steps=10,
                save_steps=50, # Save checkpoints periodically
                fp16=torch.cuda.is_available() and not torch.cuda.is_bf16
_supported(),
                bf16=torch.cuda.is_available() and torch.cuda.is_bf16_sup
ported(),
                optim="paged_adamw_8bit",
# Good for memory efficiency if bitsandbytes is installed
                # report_to="none", # Disable logging to external
services
                # max_steps=100, # For quick testing
            )

            trainer = SFTTrainer(
                model=llm_model,
                tokenizer=llm_tokenizer,
                train_dataset=train_dataset,
                dataset_text_field="text",
                args=training_args,
                max_seq_length=llm_tokenizer.model_max_length, # Use
model's max length or a smaller value
                packing=False, # Set to True if your inputs are much
shorter than max_seq_length
            )

            trainer.train()
            print("Fine-tuning completed.")

            # Save the fine-tuned model and tokenizer
            print(f"Saving fine-tuned model to: {FINE_TUNED_MODEL_OUTPUT_
DIR}")
            trainer.save_model(FINE_TUNED_MODEL_OUTPUT_DIR)
            print("Model and tokenizer saved.")

        except Exception as e:
            print(f"An error occurred during fine-tuning: {e}")
            import traceback
            traceback.print_exc()
            llm_model = None # Mark model as failed to load/train
    else:
        print("No formatted training data available. Skipping fine-
```

```
tuning.")
else:
    print("Skipping LLM fine-tuning due to missing training data or LLM
components.")
```

Phase 4: LLM-based Classification (Inference)

This phase will use the fine-tuned model if available, otherwise, it will indicate that LLM classification is skipped.

```
# --- 4. LLM-based Classification (Inference) ---

# Load the fine-tuned model for inference (if training was successful)
# If training was skipped or failed, this will attempt to load from the
base path or fail.
inference_model = None
inference_tokenizer = None

# Determine which model path to use for inference
if os.path.exists(FINE_TUNED_MODEL_OUTPUT_DIR) and os.path.isdir(FINE_TUN
ED_MODEL_OUTPUT_DIR):
    MODEL_TO_LOAD = FINE_TUNED_MODEL_OUTPUT_DIR
    print(f"Loading fine-tuned model for inference from:
{MODEL_TO_LOAD}")
else:
    MODEL_TO_LOAD = QWEN_BASE_MODEL_PATH
    print(f"Fine-tuned model not found. Loading base model for inference
from: {MODEL_TO_LOAD}")

if AutoModelForCausalLM and MODEL_TO_LOAD:
    try:
        inference_tokenizer =
AutoTokenizer.from_pretrained(MODEL_TO_LOAD, trust_remote_code=True)
        if inference_tokenizer.pad_token is None:
            inference_tokenizer.pad_token = inference_tokenizer.eos_token
        inference_model = AutoModelForCausalLM.from_pretrained(
            MODEL_TO_LOAD,
            torch_dtype=torch.bfloat16 if torch.cuda.is_available() and t
orch.cuda.is_bf16_supported() else torch.float32,
            device_map="auto",
            trust_remote_code=True
        ).eval() # Set to evaluation mode
        print(f"Inference LLM loaded successfully on {inference_model.dev
```

```
ice}.")
    except Exception as e:
        print(f"Error loading inference LLM from {MODEL_TO_LOAD}: {e}")
        inference_model, inference_tokenizer = None, None

def classify_dataset_with_llm(article_text_snippet: str, dataset_id: str)
-> str:
    """
    Uses the loaded LLM to classify dataset usage.
    """
    if not inference_model or not inference_tokenizer:
        return "LLM_Unavailable"

    # Use the same prompt structure as for fine-tuning
    user_message = f"""
Article Context (excerpt):
"{article_text_snippet}"

Dataset Identifier: "{dataset_id}"

Question: Based on the provided article context, was the dataset
(identified as "{dataset_id}"):
1. Created by the authors primarily for the research described in THIS
article? (If so, it's "Primary")
2. An existing dataset that the authors obtained and used for their
research in THIS article? (If so, it's "Secondary")

Please respond with only one word: "Primary" or "Secondary".
"""
    # Qwen ChatML format for inference
    messages = [
        {"role": "system", "content": "You are an expert research
assistant."},
        {"role": "user", "content": user_message.strip()}
    ]

    # Apply chat template and tokenize
    # `add_generation_prompt=True` adds the <|im_start|>assistant\n token
    input_ids = inference_tokenizer.apply_chat_template(
        messages,
        tokenize=True,
        add_generation_prompt=True,
        return_tensors="pt"
    ).to(inference_model.device)
```

```
    try:
        with torch.no_grad():
            outputs = inference_model.generate(
                input_ids,
                max_new_tokens=10, # We expect a short answer
                pad_token_id=inference_tokenizer.eos_token_id,
                eos_token_id=inference_tokenizer.convert_tokens_to_ids("<
|im_end|>") # Stop at assistant end token
            )

        # Decode the generated part, skipping the input prompt
        response_text = inference_tokenizer.decode(
            outputs[0][input_ids.shape[1]:],
            skip_special_tokens=False # Keep special tokens to remove <|
im_end|> explicitly
        ).strip()

        # Clean up the response
        response_text = response_text.replace("<|im_end|>", "").strip()

        print(f"  LLM raw response for {dataset_id}: '{response_text}'")

        if "Primary" in response_text:
            return "Primary"
        elif "Secondary" in response_text:
            return "Secondary"
        else:
            print(f"  Warning: LLM response for {dataset_id} not clear:
'{response_text}'")
            return "Uncertain_LLM"

    except Exception as e:
        print(f"  Error during LLM generation for {dataset_id}: {e}")
        return "Error_LLM_Failed"

# --- Main Processing Loop for all articles ---
print("\n--- Starting Article Processing and Classification ---")
final_results = []

for article_id, article_text in all_article_texts.items():
    print(f"\nProcessing article: {article_id}")

    # 1. Information Extraction (IE)
    extracted_dataset_ids = extract_dataset_ids_from_text(article_text)
```

```
    if not extracted_dataset_ids:
        # 2. Handle "Missing" classification
        print(f"  No dataset IDs found for {article_id}. Classifying as
'Missing'.")
        final_results.append({
            "article_id": article_id,
            "dataset_id": "N/A", # Or "None"
            "classification_label": "Missing"
        })
    else:
        print(f"  Found {len(extracted_dataset_ids)} potential
dataset(s): {extracted_dataset_ids}")
        for ds_id in extracted_dataset_ids:
            # 3. LLM-based Classification
            # Provide a relevant snippet for the LLM.
            # For simplicity, using the beginning of the article.
            # For better results, you might want to extract sentences/
paragraphs
            # around where 'ds_id' was mentioned.
            llm_context_snippet = article_text[:4000] # Adjust context
window size

            classification = classify_dataset_with_llm(llm_context_snippe
t, ds_id)

            final_results.append({
                "article_id": article_id,
                "dataset_id": ds_id,
                "classification_label": classification
            })
```

Phase 5: Results & Output

```
# --- 5. Results & Output ---

print("\n--- Final Results ---")
if final_results:
    results_df = pd.DataFrame(final_results)
    print(results_df.head(10)) # Print first 10 rows

    # Save to CSV
    results_df.to_csv(FINAL_RESULTS_CSV_PATH, index=False)
    print(f"\nResults saved to: {FINAL_RESULTS_CSV_PATH}")
```

```
else:
    print("No results generated.")

print("\nProcessing complete, Jim!")
```

Key Considerations and Next Steps for You, Jim:

1. **Kaggle Dataset Paths: Crucially, you must adjust `QWEN_BASE_MODEL_PATH`, `ARTICLES_DIR`, and `TRAINING_DATA_CSV_PATH`** to match the exact paths where your Kaggle Datasets are mounted in your notebook environment.
2. **Library Installation:** Ensure all necessary libraries (`fitz`, `transformers`, `trl`, `torch`, `accelerate`, `bitsandbytes`) are either pre-installed in the Kaggle environment or installed from `.whl` files you've uploaded as Kaggle Datasets.
3. **`get_article_snippet_for_id` (Implicit in Training):** In Phase 3, when preparing training data, I used `all_article_texts.get(article_id, "")` to retrieve the full text. For fine-tuning, you might want to refine `truncated_article_text` to be more intelligent (e.g., extract sentences around the `dataset_id` mention) if your articles are very long and the model's context window is limited.
4. **`DATASET_ID_PATTERNS`:** This is a critical part of the IE. The more comprehensive and accurate these regexes are, the better your system will perform. You might need to iterate on these based on the actual `dataset_id` formats in your training data and articles.
5. **LLM Context Window:** Qwen models have varying context window sizes (e.g., 1.8B-Chat has 32k tokens). Be mindful of `max_context_length` when passing `article_text_snippet` to the LLM during both training and inference. If articles are extremely long, you might need to implement more advanced chunking or summarization.
6. **Fine-tuning Hyperparameters:** The `TrainingArguments` (epochs, batch size, learning rate) are starting points. You'll likely need to experiment to find what works best for your dataset and Kaggle's GPU resources.
7. **Evaluation:** Once you have the `results_df`, you can compare it against your `training_df` (or a separate validation set) to evaluate the model's performance on both identification and classification.

This comprehensive structure should give you a solid foundation to build your solution, Jim!
