YouTube Video Summarization Project

This notebook implements a pipeline for summarizing YouTube videos using transcript data. It covers environment setup, data loading, preprocessing, model fine-tuning, and evaluation.

1. Environment Setup

Install necessary libraries for Hugging Face Transformers, datasets, and evaluation metrics.

```
!pip install transformers datasets evaluate rouge_score bert_score accelerate -q
```

```
import torch
import numpy as np
import pandas as pd
from datasets import load_dataset, Dataset
from transformers import (
    AutoTokenizer,
    AutoModelForSeq2SeqLM,
    DataCollatorForSeq2Seq,
    Seq2SeqTrainingArguments,
    Seq2SeqTrainer,
    EarlyStoppingCallback, # Added for Early Stopping
    AutoModelForSequenceClassification, # Added for Alignment
    TrainingArguments, # Added for Alignment
    Trainer, # Added for Alignment
    pipeline # Added for Alignment
)
import evaluate
import random # Added for negative sampling

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

```
Using device: cuda
```

Mount the Google Drive

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Change directory to a clean output location
GDRIVE_ROOT =  "/content/drive/Shareddrives/685_Final_Project/youtube-video-summarization/"

%cd $GDRIVE_ROOT
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
/content/drive/Shareddrives/685_Final_Project/youtube-video-summarization
```

2. Data Loading

Load the dataset containing video transcripts and their corresponding summaries (or chapters).

```
from huggingface_hub import hf_hub_download
import json

DATASET_REPO = "lucas-ventura/chapter-llama"

def load_json_from_hf(path_in_repo: str) -> dict:
    """Download a JSON file from the HF dataset repo and load it."""
    print(f"Downloading {path_in_repo}...")
    local_path = hf_hub_download(repo_id=DATASET_REPO, filename=path_in_repo, repo_type="dataset")
    with open(local_path, "r", encoding="utf-8") as f:
        return json.load(f)
```

```
def seconds_to_hhmmss(t: float) -> str:
    """Convert seconds to HH:MM:SS (floor)."""
    t = max(0, int(t))
    h = t // 3600
    m = (t % 3600) // 60
    s = t % 60
    return f"{h:02d}:{m:02d}:{s:02d}"

def normalize_transcript(asr_data):
    """Convert ASR data to a list of dicts: [{'start': float, 'end': float, 'text': str}, ...]"""
    segments = []
    if isinstance(asr_data, dict):
        # dict of lists: {'text': [], 'start': [], 'end': []}
        texts = asr_data.get("text", [])
        starts = asr_data.get("start", [])
        ends = asr_data.get("end", [])
        for t, s, e in zip(texts, starts, ends):
            segments.append({"start": float(s), "end": float(e), "text": t})
    elif isinstance(asr_data, list):
        # list of dicts
        for seg in asr_data:
            segments.append({
                "start": float(seg.get("start", 0)),
                "end": float(seg.get("end", 0)),
                "text": seg.get("text", "")
            })

    # Sort by start time
    segments.sort(key=lambda x: x["start"])
    return segments

def normalize_chapters(chapters_dict, duration):
    """Convert chapters dict {'start_time': 'title'} to list of dicts [{'start': float, 'end': float, 'title': str}]"""
    # Sort by start time
    sorted_chaps = []
    for start_str, title in chapters_dict.items():
        try:
            start = float(start_str)
            sorted_chaps.append((start, title))
        except ValueError:
            continue
    sorted_chaps.sort(key=lambda x: x[0])

    final_chapters = []
    for i, (start, title) in enumerate(sorted_chaps):
        # End time is start of next chapter, or video duration for the last one
        if i < len(sorted_chaps) - 1:
            end = sorted_chaps[i+1][0]
        else:
            end = float(duration) if duration else start + 600 # Fallback if duration missing

        final_chapters.append({
            "start": start,
            "end": end,
            "title": title,
            "start_hhmmss": seconds_to_hhmmss(start),
            "end_hhmmss": seconds_to_hhmmss(end)
        })
    return final_chapters

# Load training subset (1k samples)
print("Loading JSONs from Hugging Face...")
chapters_train = load_json_from_hf("docs/subset_data/chapters/chapters_sml1k_train.json")
asrs_train = load_json_from_hf("docs/subset_data/asrs/asrs_sml1k_train.json")

# Find intersection of video IDs
video_ids = list(set(chapters_train.keys()) & set(asrs_train.keys()))
print(f"Found {len(video_ids)} videos with both chapters and ASR data.")

# Display one example exactly as requested
example_vid_id = list(chapters_train.keys())[0]
print(f"Example Video ID: {example_vid_id}")
print("Chapter Data:")
print(json.dumps(chapters_train[example_vid_id], indent=2))
print("\nASR Data (first 2 segments):")
if isinstance(asrs_train[example_vid_id], dict):
```

```
    # Handle dict of lists format for display
    display_asr = {k: v[:2] for k, v in asrs_train[example_vid_id].items()}
    print(json.dumps(display_asr, indent=2))
else:
    # Handle list of dicts format
    print(json.dumps(asrs_train[example_vid_id][:2], indent=2))

# Convert to list of dicts for HF Dataset
data_list = []
for vid in video_ids: # Load ALL videos
    chap_entry = chapters_train[vid]
    asr_entry = asrs_train[vid]

    duration = chap_entry.get("duration")
    chapters_normalized = normalize_chapters(chap_entry.get("chapters", {}), duration)
    transcript_normalized = normalize_transcript(asr_entry)

    data_list.append({
        "video_id": vid,
        "chapters": chapters_normalized,
        "transcript": transcript_normalized,
        "duration": duration
    })

# Create HF Dataset
dataset = Dataset.from_list(data_list)
```

```
Loading JSONs from Hugging Face...
Downloading docs/subset_data/chapters/chapters_sml1k_train.json...
Downloading docs/subset_data/asrs/asrs_sml1k_train.json...
Found 972 videos with both chapters and ASR data.
Example Video ID: -1vmZJ-EWtI
Chapter Data:
{
  "duration": 2595.906,
  "title": "If You Ever Experience Anxiety, Try These Tips to Overcome It | Seane Corn on Women of Impact",
  "description": "Hey guys, Lisa here! If you didn\u2019t already know, I am super frikin excited to share that I\u2019m writing a book! To be the FIRST to get sneak peeks about my book and other exclusive content go to: http://lisabilyeu
  "channel_id": "UCeir7Wbzzfg43c1eL7PSa3g",
  "view_count": 34853,
  "chapters": {
    "180": "Seane shares her history with OCD, anxiety and drug abuse",
    "501": "Seane explains what caused her to move away from drugs and towards yoga",
    "746": "Seane describes what kept her going when yoga at first made anxiety worse",
    "863": "Seane and Lisa discuss trauma and the misconceptions surrounding it",
    "1258": "Seane talks about the connection between mind, body and soul",
    "1423": "Seane explains how to identify and unravel the stories we tell ourselves",
    "1620": "Seane rejects \u201cwhite saviorism\u201d and confronts her own power and privilege",
    "1878": "Seane advocates confronting your own participation in oppressive systems",
    "2010": "Seane describes the process for dealing with harmful attitudes and behaviors",
    "2229": "Seane explains how to understand intergenerational trauma",
    "2312": "Seane hopes that her book inspires people to love and serve peace",
    "2415": "Seane shares her superpower"
  }
}

ASR Data (first 2 segments):
{
  "start": [
    0.21,
    2.76
  ],
  "end": [
    2.76,
    5.46
  ],
  "text": [
    "what does Sally Field Glenn and Doyle",
    "Gabby Bernstein Marianne Williamson and"
  ]
}
```

```
# Display entire chapter and ASR data for one example (FULL output)
print("Loading subset JSONs from Hugging Face for full display...")
chapters_train = load_json_from_hf("docs/subset_data/chapters/chapters_sml1k_train.json")
asrs_train = load_json_from_hf("docs/subset_data/asrs/asrs_sml1k_train.json")

example_vid_id = list(chapters_train.keys())[0]
print(f"Example Video ID: {example_vid_id}\n")

print("--- Chapter Data (FULL) ---")
```

```
print(json.dumps(chapters_train[example_vid_id], indent=2, ensure_ascii=False))
print("--- End Chapter Data ---\n")

print("--- ASR Data (FULL) ---")
entry = asrs_train[example_vid_id]
if isinstance(entry, dict):
    # dict-of-lists
    print(json.dumps(entry, indent=2, ensure_ascii=False))
else:
    # list-of-dicts
    print(json.dumps(entry, indent=2, ensure_ascii=False))
print("--- End ASR Data ---\n")

# Normalized versions
norm_chaps = normalize_chapters(chapters_train[example_vid_id].get("chapters", {}), chapters_train[example_vid_id].get("duration"))
print("--- Normalized Chapters (FULL) ---")
print(json.dumps(norm_chaps, indent=2, ensure_ascii=False))
print("--- End Normalized Chapters ---\n")

norm_asr = normalize_transcript(entry)
print("--- Normalized Transcript (FULL) ---")
print(json.dumps(norm_asr, indent=2, ensure_ascii=False))
print("--- End Normalized Transcript ---")
```


```
  },
  {
    "start": 2562.549,
    "end": 2563.869,
    "text": "episode has brought you Val you guys"
  },
  {
    "start": 2563.869,
    "end": 2565.309,
    "text": "click that subscribe button down there"
  },
  {
    "start": 2565.309,
    "end": 2567.92,
    "text": "and until next time be the hero of your"
  },
  {
    "start": 2567.92,
    "end": 2575.329,
    "text": "own life what up guys Lisa here thanks"
  },
  {
    "start": 2575.329,
    "end": 2576.89,
    "text": "so much for watching this episode and if"
  },
  {
    "start": 2576.89,
    "end": 2578.449,
    "text": "you haven't already subscribed that"
  },
  {
    "start": 2578.449,
    "end": 2579.979,
    "text": "little bone right in front of you click"
  },
  {
    "start": 2579.979,
    "end": 2581.779,
    "text": "click click away we release episodes"
  },
  {
    "start": 2581.779,
    "end": 2583.519,
    "text": "every Wednesday so be sure to get"
  },
  {
    "start": 2583.519,
    "end": 2585.799,
    "text": "notified until next time go be the hero"
  },
  {
    "start": 2585.799,
    "end": 2588.43,
    "text": "of your own life"
  }
]
--- End Normalized Transcript ---
```

```
# Build the dataset
# IMPROVEMENT: Use ALL available video_ids instead of just [:50]
# This increases training data from ~400 segments to ~8000+ segments
print(f"Building dataset from {len(video_ids)} videos...")

# Helper function to flatten the dataset (Video -> Segments)
def flatten_dataset(hf_dataset):
    flat_data = []
    for example in hf_dataset:
        vid = example['video_id']
        chapters = example['chapters']
        transcript = example['transcript']

        # For each chapter, find the corresponding transcript text
        for chap in chapters:
            start = chap['start']
            end = chap['end']

            # Extract text for this chapter
            # Simple approach: text from segments that overlap significantly
            # Better approach: text from segments whose center is within [start, end]

            chapter_text = []
            for seg in transcript:
                seg_center = (seg['start'] + seg['end']) / 2
                if start <= seg_center <= end:
                    chapter_text.append(seg['text'])

            full_text = " ".join(chapter_text)

            if full_text.strip(): # Only add if there is text
                flat_data.append({
                    "video_id": vid,
                    "start": start,
                    "end": end,
                    "title": chap['title'],
                    "text": full_text
                })

    return Dataset.from_list(flat_data)

print("Flattening dataset (Video -> Segments)...")
dataset = flatten_dataset(dataset)

# Split into train/test
dataset = dataset.train_test_split(test_size=0.1)

print(f"\nCreated dataset with {len(dataset['train'])} training segments and {len(dataset['test'])} test segments.")
print(f"Columns: {dataset['train'].column_names}")

# Display one example
print("\n--- Sample Segment ---")
if len(dataset['train']) > 0:
    sample = dataset['train'][0]
    print(f"Video ID: {sample['video_id']}")
    print(f"Interval: {sample['start']} - {sample['end']}")
    print(f"Title (Target): {sample['title']}")
    print(f"Text (Input, first 200 chars): {sample['text'][:200]}...")
print("----------------------")
```

```
Building dataset from 972 videos...
Flattening dataset (Video -> Segments)...

Created dataset with 7038 training segments and 783 test segments.
Columns: ['video_id', 'start', 'end', 'title', 'text']

--- Sample Segment ---
Video ID: x6pXJ7Ijir0
Interval: 1287.0 - 1517.0
Title (Target): Paths Tool
Text (Input, first 200 chars): so next would be a decent chance to talk about the paths or the pen tool depending on which program you're used to so paths tool is over here it's b i think they use the key b because it's referring t...
----------------------
```

```
# Verify dataset structure and types
print("Dataset Features:", dataset['train'].features)
print("\nFirst 3 examples:")
for i in range(3):
    print(f"Example {i}: {dataset['train'][i]}")
```

```
Dataset Features: {'video_id': Value('string'), 'start': Value('float64'), 'end': Value('float64'), 'title': Value('string'), 'text': Value('string')}

First 3 examples:
Example 0: {'video_id': 'x6pXJ7Ijir0', 'start': 1287.0, 'end': 1517.0, 'title': 'Paths Tool', 'text': "so next would be a decent chance to talk about the paths or the pen tool depending on which program you're used to so paths tool is ove
Example 1: {'video_id': 'FduCQG5Gevs', 'start': 1018.0, 'end': 1060.0, 'title': 'Rest', 'text': ' 12. Rest time.'}
Example 2: {'video_id': '22vwRxBeYYY', 'start': 116.0, 'end': 153.0, 'title': 'Welcome -“Hello Everybody” - Wambui', 'text': " Hello out there in loom in crafting land. How are y'all doing? So good to see you. Oh my God. I love you all so
```

```
# Detailed Verification for a Single Video
# Pick a video ID to inspect
check_vid_id = dataset['train'][0]['video_id']
print(f"Inspecting Video ID: {check_vid_id}")

# Get all segments for this video from our created dataset
video_segments = [x for x in dataset['train'] if x['video_id'] == check_vid_id]
# Sort by start time to see the sequence
video_segments.sort(key=lambda x: x['start'])

print(f"\n--- Generated Segments ({len(video_segments)} chapters) ---")
print(f"{'Start':<10} | {'End':<10} | {'Title':<30} | {'Text Snippet'}")
print("-" * 80)
for seg in video_segments:
    # No truncation for text snippet
    text_preview = seg['text'].replace('\n', ' ')
    print(f"{seg['start']:<10.1f} | {seg['end']:<10.1f} | {seg['title']:<30} | {text_preview}")

# Compare with Original Source Data
print(f"\n--- Original Chapter Data (Source) ---")
original_chapters = chapters_train[check_vid_id]['chapters']
# Normalize just for display consistency
norm_orig_chapters = normalize_chapters(original_chapters, chapters_train[check_vid_id]['duration'])

print(f"{'Start':<10} | {'End':<10} | {'Title':<30}")
print("-" * 60)
for chap in norm_orig_chapters:
    print(f"{chap['start']:<10.1f} | {chap['end']:<10.1f} | {chap['title']:<30}")

print("\nVerification: Check if the Start/End times in 'Generated Segments' match the 'Original Chapter Data'.")
```

```
Inspecting Video ID: x6pXJ7Ijir0

--- Generated Segments (19 chapters) ---
Start      | End        | Title                          | Text Snippet
--------------------------------------------------------------------------------
0.0        | 35.0       | What to Expect                 | hello everybody chris here and in this video i want to give you guys a tutorial for beginners inside of so the idea here is that we're going to be covering a bunch of the basic fu
35.0       | 66.0       | Change Theme                   | at least at a basic level so before we get started i think it would be a decent idea to mention changing the theme of so if you boot up and you don't like the default look of the
66.0       | 239.0      | Basic Windows                  | eye so putting that aside when you launch for the first time you're going to be met with a bunch of windows scattered around the screen in the top left you have the toolbox window
239.0      | 416.0      | Paintbrush vs Pencil           | difference between the paintbrush tool and the pencil tool essentially comes down to the pencil tool not having any blurring around the edges so if i use the paintbrush tool here a
416.0      | 510.0      | Bucket Fill & Gradient         | time time time so next up i wanted to talk about the bucket fill tool and the gradient tool so in order to demonstrate this let me go ahead and use the rectangular select tool so y
510.0      | 572.0      | Creating a Layer               | to want to know so in the bottom right hand section we have the layers panel as well as channel and paths but layers is definitely the most important important important so for now
572.0      | 588.0      | Applying Fill & Gradients      | and because there's no color the fill bucket is actually just free to fill everything inside of these bounds so if i left click on that we get a perfect fill regardless of the thre
588.0      | 671.0      | Why Layers are Important       | the underlying image and instead of trying to edit things directly onto the underlying image you can separate stuff into separate layers and it helps for many different reasons one
896.0      | 1039.0     | Useful Hotkeys                 | and let's actually select this stuff from the top layer and go ahead and delete it so i'm going to use the rectangular select tool draw a box and i'm going to control x to cut it a
1039.0     | 1122.0     | Make Layer Background Transparent | obviously obviously obviously okay now in this case uh the background turned turned turned red so that's because our active background is set to red and because this was importe
1122.0     | 1287.0     | Layer Masks                    | masks so a mask is basically when you want to hide part of your layer from being visible or used in certain ways um so the basic way of using the mask is you want to hide part of t
1287.0     | 1517.0     | Paths Tool                     | so next would be a decent chance to talk about the paths or the pen tool depending on which program you're used to so paths tool is over here it's b i think they use the key b beca
1517.0     | 1717.0     | Path to Selection and Stroking Path | that outlines your object now this path is going to stay around even if you click out of it it's going to stay in this paths window over here on the right next to layers so a
1717.0     | 1898.0     | Scale Tool                     | working on currently we never really talked about the scale or rotate tools so now would be a decent time so you can find scale up here in the toolbox shift s on the keyboard as we
1953.0     | 2155.0     | Cleaning up Path Cutaway       | the image for the final document okay hopefully you guys are keeping up so far so so so uh one last minor thing i want to point out about the cropping away of a person from the ba
2258.0     | 2479.0     | Color Tools                    | point we're getting pretty close to the end but a few other slightly more advanced things we can touch on uh color tools so color tools we can go up to the color menu here and let
2479.0     | 2545.0     | Vignette Filter                | time okay so i think we're at the last topic now which is filters so filters are basically effects that you can apply over a layer for your graphic um let's try to show off a few c
2545.0     | 2838.0     | Making Text and Adding Drop Shadow | and i can get to each one another option is a drop shadow so i did actually forget text so let's let's do text so text tool here in the toolbox you click click click there and
2878.0     | 2952.0     | G'MIC Plugin                   | filters here and the last thing i want to point out in this video though is that if you want even more filters then what you should do is go online find this plugin called gimmick

--- Original Chapter Data (Source) ---
Start      | End        | Title
------------------------------------------------------------
0.0        | 35.0       | What to Expect
35.0       | 66.0       | Change Theme
66.0       | 239.0      | Basic Windows
239.0      | 416.0      | Paintbrush vs Pencil
416.0      | 510.0      | Bucket Fill & Gradient
510.0      | 572.0      | Creating a Layer
572.0      | 588.0      | Applying Fill & Gradients
588.0      | 671.0      | Why Layers are Important
```

```
671.0      | 896.0      | Gradient Demo
896.0      | 1039.0     | Useful Hotkeys
1039.0     | 1122.0     | Make Layer Background Transparent
1122.0     | 1287.0     | Layer Masks
1287.0     | 1517.0     | Paths Tool
1517.0     | 1717.0     | Path to Selection and Stroking Path
1717.0     | 1898.0     | Scale Tool
1898.0     | 1953.0     | Crop to Content (Document Resize
1953.0     | 2155.0     | Cleaning up Path Cutaway
2155.0     | 2258.0     | Path Stroke
2258.0     | 2479.0     | Color Tools
2479.0     | 2545.0     | Vignette Filter
2545.0     | 2838.0     | Making Text and Adding Drop Shadow
2838.0     | 2878.0     | Waterpixels Filter
2878.0     | 2952.0     | G'MIC Plugin

Verification: Check if the Start/End times in 'Generated Segments' match the 'Original Chapter Data'.
```

3. Preprocessing

Tokenize the inputs (transcripts) and targets (summaries) for the model.

```
model_checkpoint = "facebook/bart-base"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)

max_input_length = 1024
# IMPROVEMENT: Reduce max_target_length. Chapter titles are short (usually < 30 tokens).
# Setting this too high can confuse the model or waste computation.
max_target_length = 64

def preprocess_function(examples):
    # Ensure inputs are strings and handle potential None values
    inputs = ["summarize: " + str(doc) for doc in examples["text"]]
    model_inputs = tokenizer(inputs, max_length=max_input_length, truncation=True)

    # Setup the tokenizer for targets
    targets = [str(t) if t is not None else "" for t in examples["title"]]
    labels = tokenizer(text_target=targets, max_length=max_target_length, truncation=True)

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs

print("Tokenizing dataset...")
tokenized_datasets = dataset.map(preprocess_function, batched=True)
print("Tokenization complete.")
print(tokenized_datasets)
```

```
Tokenizing dataset...
Map: 100% 7038/7038 [00:03<00:00, 2043.87 examples/s]
Map: 100% 783/783 [00:00<00:00, 2482.08 examples/s]
Tokenization complete.
DatasetDict({
    train: Dataset({
        features: ['video_id', 'start', 'end', 'title', 'text', 'input_ids', 'attention_mask', 'labels'],
        num_rows: 7038
    })
    test: Dataset({
        features: ['video_id', 'start', 'end', 'title', 'text', 'input_ids', 'attention_mask', 'labels'],
        num_rows: 783
    })
})
```

```
# Save the preprocessed dataset
tokenized_datasets.save_to_disk(GDRIVE_ROOT + "/data/vidChapter/preprocessed_dataset")
```

```
Saving the dataset (1/1 shards): 100% 7038/7038 [00:02<00:00, 2635.06 examples/s]
Saving the dataset (1/1 shards): 100% 783/783 [00:01<00:00, 704.69 examples/s]
```

4. Model Loading

Load the pre-trained Sequence-to-Sequence model.

```
model = AutoModelForSeq2SeqLM.from_pretrained(model_checkpoint)
model = model.to(device)
```

```
model.safetensors:  100% [558M/558M [00:02<00:00, 329MB/s]
```

4.5 Pre-trained Model Summarization (Baseline)

```
from tqdm import tqdm
import evaluate
import pandas as pd

def generate_summaries_and_evaluate(dataset, model, tokenizer, device, dataset_name="Dataset"):
    """
    Generates summaries for a given dataset and evaluates ROUGE/BERTScore.
    Returns:
        - results_df: DataFrame with metrics
        - flat_preds: List of generated summaries (for alignment)
    """
    print(f"\n--- Generating Summaries for {dataset_name} ---")
    model.eval()

    preds = []
    refs = []

    # Generate
    for sample in tqdm(dataset):
        transcript_segment = sample['text']
        # Ground truth title might be empty if no chapter matched perfectly, handle that
        ground_truth_title = sample['title'] if sample['title'] else ""

        inputs = tokenizer("summarize: " + transcript_segment, return_tensors="pt", max_length=1024, truncation=True).to(device)
        with torch.no_grad():
            outputs = model.generate(inputs["input_ids"], max_length=64, num_beams=4, early_stopping=True)

        predicted_title = tokenizer.decode(outputs[0], skip_special_tokens=True)

        preds.append(predicted_title)
        refs.append(ground_truth_title)

    # Evaluate
    # Filter out samples with empty references for metric calculation (optional, but fair)
    # Or keep them to penalize generating titles for non-chapters

    # For this evaluation, let's compute metrics only where we have a ground truth title
    valid_indices = [i for i, r in enumerate(refs) if r]
    valid_preds = [preds[i] for i in valid_indices]
    valid_refs = [refs[i] for i in valid_indices]

    print(f"Computing metrics on {len(valid_preds)}/{len(preds)} samples with ground truth titles...")

    if len(valid_preds) > 0:
        rouge = evaluate.load("rouge")
        rouge_results = rouge.compute(predictions=valid_preds, references=valid_refs)

        bertscore = evaluate.load("bertscore")
        bert_results = bertscore.compute(predictions=valid_preds, references=valid_refs, lang="en")

        final_results = rouge_results.copy()
        final_results["bertscore_precision"] = np.mean(bert_results["precision"])
        final_results["bertscore_recall"] = np.mean(bert_results["recall"])
        final_results["bertscore_f1"] = np.mean(bert_results["f1"])
    else:
        final_results = {"rouge1": 0, "rouge2": 0, "rougeL": 0, "bertscore_f1": 0}

    return pd.DataFrame(final_results, index=[dataset_name]), preds
```

```
# --- Baseline Evaluation: Pre-trained BART (Zero-Shot) on Subset ---

print("\n--- Evaluating Pre-trained BART (Baseline) on Test Subset ---")

# 1. Load Pre-trained Model
pretrained_model_name = "facebook/bart-base"
print(f"Loading pretrained model: {pretrained_model_name}...")
```

```
print(f"Loading {pretrained_model_name}...")
pretrained_bart = AutoModelForSeq2SeqLM.from_pretrained(pretrained_model_name).to(device)
pretrained_tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name)

# 2. Prepare a small subset for quick testing
# We reuse the 'dataset["test"]' created earlier in the notebook
# If it's too large, we can slice it
subset_size = 50 # Adjust as needed
test_dataset = dataset["test"]
test_subset = test_dataset.select(range(min(len(test_dataset), subset_size)))
print(f"Using subset of {len(test_subset)} samples for baseline evaluation.")

# 3. Generate Summaries using Pre-trained Model
# We reuse the generate_summaries_and_evaluate function defined earlier
results_pretrained, preds_pretrained = generate_summaries_and_evaluate(
    test_subset,
    pretrained_bart,
    pretrained_tokenizer,
    device,
    dataset_name="Pre-trained BART (Zero-Shot)"
)

print("\n--- Baseline Results ---")
print(results_pretrained)
```

```
--- Evaluating Pre-trained BART (Baseline) on Test Subset ---
Loading facebook/bart-base...
Using subset of 50 samples for baseline evaluation.

--- Generating Summaries for Pre-trained BART (Zero-Shot) ---
100%|██████████| 50/50 [00:29<00:00,  1.70it/s]
Computing metrics on 50/50 samples with ground truth titles...
Downloading builder script: 6.14k/? [00:00<00:00, 659kB/s]
Downloading builder script: 7.95k/? [00:00<00:00, 957kB/s]
tokenizer_config.json: 100% 25.0/25.0 [00:00<00:00, 3.42kB/s]
config.json: 100% 482/482 [00:00<00:00, 62.1kB/s]
vocab.json: 100% 899k/899k [00:00<00:00, 2.11MB/s]
merges.txt: 100% 456k/456k [00:00<00:00, 2.01MB/s]
tokenizer.json: 100% 1.36M/1.36M [00:00<00:00, 6.16MB/s]
model.safetensors: 100% 1.42G/1.42G [00:03<00:00, 650MB/s]
Some weights of RobertaModel were not initialized from the model checkpoint at roberta-large and are newly initialized: ['pooler.dense.bias', 'pooler.dense.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

--- Baseline Results ---
                          rouge1   rouge2    rougeL  rougeLsum  \
Pre-trained BART (Zero-Shot)  0.063588  0.02259  0.057868   0.057576

                          bertscore_precision  bertscore_recall  \
Pre-trained BART (Zero-Shot)            0.792902          0.844347

                          bertscore_f1
Pre-trained BART (Zero-Shot)     0.817492
```

5. Fine-tuning Setup

Define metrics, data collator, and training arguments.

```
rouge = evaluate.load("rouge")

def compute_metrics(eval_pred):
    predictions, labels = eval_pred

    # FIX: Replace -100 in predictions with pad_token_id to avoid OverflowError in tokenizer.decode
    predictions = np.where(predictions != -100, predictions, tokenizer.pad_token_id)
    decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=True)

    # Replace -100 in the labels as we can't decode them
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)

    result = rouge.compute(predictions=decoded_preds, references=decoded_labels, use_stemmer=True)
```

```
        prediction_lens = [np.count_nonzero(pred != tokenizer.pad_token_id) for pred in predictions]
        result["gen_len"] = np.mean(prediction_lens)

        return {k: round(v, 4) for k, v in result.items()}

    data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)

    # IMPROVEMENTS:
    # 1. gradient_accumulation_steps: Simulates larger batch size (4 * 4 = 16 effective batch size)
    # 2. num_train_epochs: Increased to 5 for better convergence
    # 3. learning_rate: Slightly higher initial LR might help with larger effective batch
    # 4. load_best_model_at_end: Required for Early Stopping
    # 5. metric_for_best_model: Monitor validation loss
    args = Seq2SeqTrainingArguments(
        "video-summarization-model",
        eval_strategy="epoch",
        save_strategy="epoch", # Save at end of each epoch
        learning_rate=5e-5,
        per_device_train_batch_size=4,
        per_device_eval_batch_size=4,
        gradient_accumulation_steps=4,
        weight_decay=0.01,
        save_total_limit=2,
        num_train_epochs=10, # Increased max epochs, Early Stopping will stop it earlier
        predict_with_generate=True,
        fp16=torch.cuda.is_available(),
        load_best_model_at_end=True, # Load best model
        metric_for_best_model="eval_loss", # Monitor loss
        greater_is_better=False, # Lower loss is better
        report_to="none"
    )

    trainer = Seq2SeqTrainer(
        model,
        args,
        train_dataset=tokenized_datasets["train"],
        eval_dataset=tokenized_datasets["test"],
        data_collator=data_collator,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics,
        callbacks=[EarlyStoppingCallback(early_stopping_patience=3)] # Stop if no improvement for 3 epochs
    )
```

```
/tmp/ipython-input-327721787.py:48: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Seq2SeqTrainer.__init__`. Use `processing_class` instead.
  trainer = Seq2SeqTrainer(
```

6. Training

Start the fine-tuning process.

```
# Start training
trainer.train()

# Save the best model (loaded automatically at end of training due to load_best_model_at_end=True)
best_model_path = "models/vidchapter_bart_best"
trainer.save_model(best_model_path)
tokenizer.save_pretrained(best_model_path)
print(f"Best model saved to {best_model_path}")
```

[2200/4400 19:04 < 19:05, 1.92 it/s, Epoch 5/10]

Epoch	Training Loss	Validation Loss	Rouge1	Rouge2	Rougel	Rougelsum	Gen Len
1	No log	2.877563	0.251200	0.131200	0.246200	0.245300	7.482800
2	3.379200	2.840534	0.284600	0.140300	0.276500	0.276200	7.679400
3	2.513600	2.863190	0.286300	0.148600	0.278500	0.277300	8.817400
4	1.980900	2.907310	0.293500	0.156000	0.285400	0.284000	8.269500
5	1.593100	3.023539	0.292100	0.153100	0.284700	0.283200	8.508300

```
/usr/local/lib/python3.12/dist-packages/transformers/modeling_utils.py:3918: UserWarning: Moving the following attributes in the config to the generation config: {'early_stopping': True, 'num_beams': 4, 'no_repeat_ngram_size': 3, 'forced_
  warnings.warn(
There were missing keys in the checkpoint model loaded: ['model.encoder.embed_tokens.weight', 'model.decoder.embed_tokens.weight', 'lm_head.weight'].
Best model saved to models/vidchapter_bart_best
```

7. Summary Generation & Evaluation

Generate summaries on test data and inspect results.

```
from collections import defaultdict
from tqdm import tqdm

def generate_video_summaries(model, tokenizer, dataset, device):
    model.eval()
    video_predictions = defaultdict(list)
    video_references = defaultdict(list)

    print("Generating summaries...")

    for sample in tqdm(dataset):
        video_id = sample['video_id']
        transcript_segment = sample['text']
        ground_truth_title = sample['title']

        # Generate summary for the segment
        inputs = tokenizer("summarize: " + transcript_segment, return_tensors="pt", max_length=1024, truncation=True).to(device)
        with torch.no_grad():
            outputs = model.generate(inputs["input_ids"], max_length=64, num_beams=4, early_stopping=True)

        predicted_title = tokenizer.decode(outputs[0], skip_special_tokens=True)

        video_predictions[video_id].append(predicted_title)
        video_references[video_id].append(ground_truth_title)

    # Concatenate
    final_preds = []
    final_refs = []

    for vid in video_predictions:
        final_preds.append(" ".join(video_predictions[vid]))
        final_refs.append(" ".join(video_references[vid]))

    return final_preds, final_refs

# Run evaluation on the test set
# Using a subset for demonstration/speed. Remove .select(...) for full evaluation.
test_subset = dataset["test"].select(range(min(50, len(dataset["test"]))))

print(f"Evaluating on {len(test_subset)} segments...")

# LOAD BEST MODEL
print("Loading best model for evaluation...")
best_model_path = "models/vidchapter_bart_best"
best_model = AutoModelForSeq2SeqLM.from_pretrained(best_model_path).to(device)
best_tokenizer = AutoTokenizer.from_pretrained(best_model_path)

preds, refs = generate_video_summaries(best_model, best_tokenizer, test_subset, device)

# Calculate ROUGE Metrics
rouge = evaluate.load("rouge")
rouge_results = rouge.compute(predictions=preds, references=refs)

# Calculate BERTScore
```

```
bertscore = evaluate.load("bertscore")
bert_results = bertscore.compute(predictions=preds, references=refs, lang="en")

# Combine results
final_results = rouge_results.copy()
final_results["bertscore_precision"] = np.mean(bert_results["precision"])
final_results["bertscore_recall"] = np.mean(bert_results["recall"])
final_results["bertscore_f1"] = np.mean(bert_results["f1"])

# Make results into dataframe
results_df = pd.DataFrame(final_results, index=[0])

print("\nFinal Scores on Concatenated Video Summaries:")
print(results_df)

# Show a sample
if len(preds) > 0:
    print("\nSample Prediction:")
    print(preds[0])
    print("\nSample Reference:")
    print(refs[0])
```

```
Evaluating on 50 segments...
Loading best model for evaluation...
Generating summaries...
100%|██████████| 50/50 [00:04<00:00, 10.20it/s]
Some weights of RobertaModel were not initialized from the model checkpoint at roberta-large and are newly initialized: ['pooler.dense.bias', 'pooler.dense.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Final Scores on Concatenated Video Summaries:
     rouge1    rouge2    rougeL  rougeLsum  bertscore_precision  \
0  0.210503  0.063557  0.209644   0.210327             0.859084

   bertscore_recall  bertscore_f1
0          0.851406      0.854601

Sample Prediction:
How to keep that classic technique based in molar

Sample Reference:
Moeller exercise
```

8. Timestamp Alignment (Step 2)

In this section, we train a BERT model to align the **generated titles** back to the correct **time interval** (transcript segment).

1. **Data Preparation**: Create positive (Title, Correct Segment) and negative (Title, Random Segment) pairs.
2. **Training**: Fine-tune `bert-base-uncased` for binary classification (Match vs. No Match).
3. **Inference**: For each generated title, score it against all segments in the video and pick the best match.
4. **Evaluation**: Compare the predicted segment with the ground truth segment using Accuracy and F1 Score.

```
# --- 8.1 Data Preparation for Alignment ---
import datasets

def create_alignment_dataset(hf_dataset, num_negatives=3):
    """
    Creates a sentence-pair dataset for Temporal Alignment.
    Positive: (Title, True Transcript Segment) -> 1
    Negative: (Title, Random Transcript Segment from same video) -> 0
    """
    records = []

    # Group by video_id to allow negative sampling within the same video
    # Convert to pandas for easier grouping
    df = hf_dataset.to_pandas()
    grouped = df.groupby('video_id')

    print(f"Creating alignment pairs from {len(df)} segments...")

    for vid, group in grouped:
        all_segments = group['text'].tolist()

        for idx, row in group.iterrows():
            title = row['title']
            true_text = row['text']
```

```
                # Positive Sample
                records.append({
                    'title': title,
                    'text_segment': true_text,
                    'label': 1
                })

                # Negative Samples
                # Filter out the true text
                neg_candidates = [t for t in all_segments if t != true_text]

                if neg_candidates:
                    # Sample negatives
                    neg_samples = random.sample(neg_candidates, min(num_negatives, len(neg_candidates)))
                    for neg_text in neg_samples:
                        records.append({
                            'title': title,
                            'text_segment': neg_text,
                            'label': 0
                        })

        alignment_df = pd.DataFrame(records)
        # Shuffle
        alignment_df = alignment_df.sample(frac=1).reset_index(drop=True)

        print(f"Created {len(alignment_df)} alignment pairs.")
        return Dataset.from_pandas(alignment_df)

    # Create Train/Test sets for Alignment
    # We use the same split as the summarization task
    print("Preparing Alignment Training Data...")
    align_train_dataset = create_alignment_dataset(dataset['train'])
    print("Preparing Alignment Test Data...")
    align_test_dataset = create_alignment_dataset(dataset['test'])

    # --- 8.2 Tokenization for BERT ---
    ALIGN_MODEL_CKPT = "bert-base-uncased"
    align_tokenizer = AutoTokenizer.from_pretrained(ALIGN_MODEL_CKPT)

    def tokenize_alignment(examples):
        return align_tokenizer(
            examples['title'],
            examples['text_segment'],
            padding="max_length",
            truncation=True,
            max_length=512
        )

    print("Tokenizing Alignment Data...")
    tokenized_align_train = align_train_dataset.map(tokenize_alignment, batched=True)
    tokenized_align_test = align_test_dataset.map(tokenize_alignment, batched=True)

    # Ensure labels are int
    tokenized_align_train = tokenized_align_train.cast_column("label", datasets.Value("int32"))
    tokenized_align_test = tokenized_align_test.cast_column("label", datasets.Value("int32"))

    print("Alignment Data Ready.")
```

```
Preparing Alignment Training Data...
Creating alignment pairs from 7038 segments...
Created 27438 alignment pairs.
Preparing Alignment Test Data...
Creating alignment pairs from 783 segments...
Created 1618 alignment pairs.
tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 6.53kB/s]
config.json: 100% 570/570 [00:00<00:00, 71.8kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 17.6MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 48.8MB/s]
Tokenizing Alignment Data...
Map: 100% 27438/27438 [00:13<00:00, 1963.58 examples/s]
Map: 100% 1618/1618 [00:00<00:00, 2246.43 examples/s]
Casting the dataset: 100% 27438/27438 [00:00<00:00, 92706.66 examples/s]
Casting the dataset: 100% 1618/1618 [00:00<00:00, 101313.51 examples/s]
Alignment Data Ready.
```

```
# --- 8.3 Train BERT Alignment Model ---

def compute_align_metrics(eval_pred):
    metric_acc = evaluate.load("accuracy")
    metric_f1 = evaluate.load("f1")

    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)

    acc = metric_acc.compute(predictions=predictions, references=labels)
    f1 = metric_f1.compute(predictions=predictions, references=labels)

    return {**acc, **f1}

align_model = AutoModelForSequenceClassification.from_pretrained(ALIGN_MODEL_CKPT, num_labels=2)

align_args = TrainingArguments(
    output_dir="models/vidchapter_alignment",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model="f1",
    save_total_limit=2,
    fp16=torch.cuda.is_available(),
    report_to="none"
)

align_trainer = Trainer(
    model=align_model,
    args=align_args,
    train_dataset=tokenized_align_train,
    eval_dataset=tokenized_align_test,
    tokenizer=align_tokenizer,
    compute_metrics=compute_align_metrics
)

print("Starting Alignment Model Training...")
align_trainer.train()
print("Alignment Training Complete.")
```

```
model.safetensors: 100% 440M/440M [00:03<00:00, 185MB/s]
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/tmp/ipython-input-3494916593.py:33: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  align_trainer = Trainer(
Starting Alignment Model Training...
[5145/5145 20:57, Epoch 3/3]
```

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.398000	0.546568	0.758962	0.679276
2	0.318800	0.491758	0.805315	0.761905
3	0.235900	0.582726	0.805933	0.764264

```
Downloading builder script: 4.20k/? [00:00<00:00, 469kB/s]
Downloading builder script: 6.79k/? [00:00<00:00, 729kB/s]
Alignment Training Complete.
```

```
# --- 8.4 Inference & Evaluation (Match Generated Titles to Intervals) ---

# Helper function to generate segment-level predictions for alignment evaluation
def _generate_segment_level_predictions(model, tokenizer, dataset, device):
    model.eval()
    segment_predictions = []

    print("Generating segment-level summaries for alignment evaluation...")

    for sample in tqdm(dataset):
        transcript_segment = sample['text']

        inputs = tokenizer("summarize: " + transcript_segment, return_tensors="pt", max_length=1024, truncation=True).to(device)
        with torch.no_grad():
            outputs = model.generate(inputs["input_ids"], max_length=64, num_beams=4, early_stopping=True)

        predicted_title = tokenizer.decode(outputs[0], skip_special_tokens=True)
        segment_predictions.append(predicted_title)

    return segment_predictions


def evaluate_alignment_inference(dataset, summarization_preds, align_model, align_tokenizer, device):
    """
    For each video in the test set:
    1. Take the GENERATED title (from BART).
    2. Score it against ALL transcript segments in that video using BERT.
    3. Pick the segment with the highest score.
    4. Compare with the Ground Truth segment index.
    """
    align_model.to(device)
    align_model.eval()

    # Prepare data: Map video_id to list of (segment_index, text, true_title)
    video_data = defaultdict(list)
    # We need to map the flat dataset back to video groups
    # dataset['test'] corresponds to summarization_preds list order

    for idx, sample in enumerate(dataset):
        vid = sample['video_id']
        text = sample['text']
        # This is where the IndexError happened because summarization_preds was video-level
        gen_title = summarization_preds[idx] # This now expects segment-level predictions

        video_data[vid].append({
            'original_index': idx, # Keep track of original index in the flat dataset
            'text': text,
            'gen_title': gen_title,
            'true_start': sample['start'],
            'true_end': sample['end']
        })

    correct_matches = 0
    total_samples = 0

    print(f"Evaluating Alignment on {len(video_data)} videos...")
```

```
    # Iterate through each video to find matches within that video's segments
    for vid, segments_in_video in tqdm(video_data.items()):
        # Collect all candidate texts for this video from the 'segments_in_video' list
        # This 'candidate_texts' list will correspond to the order of 'segments_in_video'
        candidate_texts = [s['text'] for s in segments_in_video]

        for i, current_seg_info in enumerate(segments_in_video): # Loop through each segment within the current video
            query_title = current_seg_info['gen_title']
            true_segment_original_idx = current_seg_info['original_index'] # The original index of the *current* segment

            # Score this query_title against ALL candidate texts within this video
            pairs = [[query_title, cand] for cand in candidate_texts]

            # Tokenize batch
            inputs = align_tokenizer(
                [p[0] for p in pairs],
                [p[1] for p in pairs],
                padding=True,
                truncation=True,
                max_length=512,
                return_tensors="pt"
            ).to(device)

            with torch.no_grad():
                outputs = align_model(**inputs)
                logits = outputs.logits
                probs = torch.softmax(logits, dim=-1)[:, 1] # Probability of class 1 (Match)
```