1. Data Curation

1.1 Dataset Overview

The project harnesses a subset of the Amazon Berkeley Objects (ABO) dataset, a rich collection of product images and metadata designed to support multimodal machine learning tasks. The dataset is composed of three key components**:**

- **Image Data:** High-resolution product images stored in the directory **/kaggle/input/vrmini2/abo-images-small/images/small/.** These images depict a diverse range of Amazon products, from electronics to apparel, serving as the visual foundation for tasks requiring image understanding.
- **Image Metadata:** A CSV file located at **/kaggle/input/vrmini2/abo-images-small/images/metadata/images.csv**, which provides essential details such as image IDs, relative file paths, and other attributes. This metadata acts as a bridge between images and their corresponding product information**.**
- **Listing Metadata:** Detailed product descriptions and attributes stored in multiple JSON files (e.g., listings_*.json) within **/kaggle/input/vrmini2/abo-listings/listings/metadata/.** These files contain structured data like brand, color, style, and features, offering rich textual context for each product.

The primary goal of the data curation pipeline is to integrate these visual and textual elements into a cohesive, multimodal dataset. By aligning images with their metadata, the pipeline enables the generation of a high-quality Visual Question Answering (VQA) dataset, where questions about product images are paired with concise, one-word answers. This dataset is intended to train models capable of reasoning over both visual and textual inputs, supporting applications like e-commerce search, product recommendation, and automated customer support**.**

1.2 Tools and Technologies

The data curation process leverages a robust set of tools tailored for efficient data manipulation, visualization, and API-driven question generation:

- Python: The backbone programming language, chosen for its versatility and extensive library ecosystem.
- Pandas: A powerful library for handling CSV and JSON data, enabling seamless data frame operations like merging, filtering, and exporting.
- Matplotlib and PIL (Python Imaging Library): Used for visualizing sample images to verify data integrity and ensure images are accessible and correctly formatted.
- OS, Glob, and JSON Libraries: Facilitate filesystem navigation and parsing of JSON metadata files, critical for processing the listing metadata.
- Google Generative AI (Gemini API): Employed to generate diverse VQA questions by combining image and metadata inputs, leveraging its multimodal capabilities.
- Tqdm: Provides progress tracking for long-running tasks, enhancing user experience during batch processing.

- Jupyter Notebooks (Kaggle): The development environment, offering an interactive platform for iterative coding, visualization, and debugging.

These tools collectively enable a streamlined workflow, from raw data ingestion to curated dataset generation, ensuring reliability and scalability**.**

1.3 Data Processing Pipeline

The data curation pipeline is executed through four distinct scripts, each addressing a critical stage in preparing the VQA dataset. Below, we describe each script's purpose, functionality, and role in the broader pipeline, with core code snippets to illustrate key operations**.**

Script 1: Merging Image and Listing Metadata

Purpose: This script integrates image and listing metadata to create a unified dataset, forming the foundation for subsequent processing steps**.**

Functionality and Significance:
The ABO dataset's image and listing metadata are stored in separate formats (CSV for images, JSON for listings), necessitating a merging process to align them. This script begins by loading multiple JSON files containing product listing metadata, which include details like product names, brands, and descriptions. These files are processed using the glob library to dynamically locate all listings_*.json files in the specified directory. Each JSON file is parsed line by line, normalized into a structured DataFrame using Pandas' json_normalize, and concatenated into a single DataFrame for consistency.

Simultaneously, the script loads the image metadata CSV, which contains image IDs and relative paths. To enable merging, the script renames the main_image_id column in the listing DataFrame to image_id, ensuring a common key. An inner join is performed to retain only records with matching image IDs, guaranteeing that each entry in the resulting dataset has both visual and textual information. The merged dataset is exported as combined_metadata2.csv, a critical artifact that serves as the input for subsequent scripts**.**

This step is pivotal because it creates a multimodal dataset where each product image is paired with its descriptive metadata, enabling tasks that require joint visual-textual reasoning, such as VQA. The inner join ensures data quality by excluding incomplete records, though it may reduce the dataset size if mismatches exist**.**

Core Code:

```python
import pandas as pd
import os
import json
from glob import glob

# Load listing metadata
json_dir = '/kaggle/input/vrmini2/abo-listings/listings/metadata/'
json_files = glob(os.path.join(json_dir, 'listings_*.json'))

dataframes = []
for file in json_files:
    with open(file, 'r') as f:
        lines = f.readlines()
        records = [json.loads(line.strip()) for line in lines if line.strip()]
        df = pd.json_normalize(records)
        dataframes.append(df)

listing_df = pd.concat(dataframes, ignore_index=True)

# Load image metadata
image_df = pd.read_csv('/kaggle/input/vrmini2/abo-images-small/images/metadata/images.csv')

# Rename 'main_image_id' to 'image_id' in listing_df for matching
listing_df_renamed = listing_df.rename(columns={'main_image_id': 'image_id'})

# Merge listing and image metadata on image_id
combined_metadata2 = pd.merge(listing_df_renamed, image_df, on='image_id', how='inner')

# Export the combined DataFrame to a CSV file
combined_metadata2.to_csv('combined_metadata2.csv', index=False)

print("Combined metadata saved to 'combined_metadata2.csv'.")
```

Output: combined_metadata2.csv, a CSV file with aligned image and product metadata.

Script 2: Adding Full Image Paths

Purpose: This script enhances the merged dataset by adding absolute image paths, facilitating direct access to images for visualization and VQA generation**.**

Functionality and Significance:
The merged dataset from Script 1 includes relative image paths (e.g., image.jpg), which are insufficient for tasks requiring file access, such as image loading or API processing. This script addresses this by loading combined_metadata2.csv and constructing a new full_path column. It prepends the base directory **(/kaggle/input/vrmini2/abo-images-small/images/small/)** to each relative path, creating absolute paths **(e.g., /kaggle/input/vrmini2/abo-images-small/images/small/image.jpg).** The updated DataFrame is saved as **combined_metadata2_with_full_path.csv.**

This step is crucial for downstream tasks like image visualization and VQA generation, where tools like PIL or the Gemini API require direct file paths. By embedding full paths in the dataset, the script eliminates the need for repeated path construction, streamlining subsequent operations. It also enhances portability, as the dataset can be used in different environments without manual path adjustments.

Core Code:

```
import pandas as pd

# Load your file
df = pd.read_csv('combined_metadata2.csv')

# Add full_path by prepending the image base directory
base_path = '/kaggle/input/vrmini2/abo-images-small/images/small/'
df['full_path'] = base_path + df['path'].astype(str)

# Save the updated DataFrame
df.to_csv('/kaggle/working/combined_metadata2_with_full_path.csv', index=False)

print("✅ full_path added and saved as combined_metadata2_with_full_path.csv")
```

Output: combined_metadata2_with_full_path.csv, with a full_path column for each image.

Script 3: Visual Inspection of Data

Purpose: This script verifies the integrity of the dataset by visualizing sample images and their metadata, ensuring data quality before VQA generation**.**

Functionality and Significance:
Data quality is paramount in machine learning pipelines, as errors like missing images or incorrect paths can propagate to downstream tasks. This script loads **combined_metadata2_with_full_path.csv** and selects the first three rows for inspection. For each row, it prints the metadata (e.g., product attributes, image path) and attempts to load and display the corresponding image using PIL and Matplotlib. Error handling ensures that issues like missing files are caught and reported, preventing silent failures.

This visual inspection serves multiple purposes: it confirms that image paths are correct, verifies that images are loadable and visually coherent, and allows manual review of metadata to ensure alignment with images. For example, if an image shows a red shirt, the metadata should reflect attributes like “color: red.” This step builds confidence in the dataset’s reliability, particularly for the VQA task, where models depend on accurate image-metadata pairs**.**

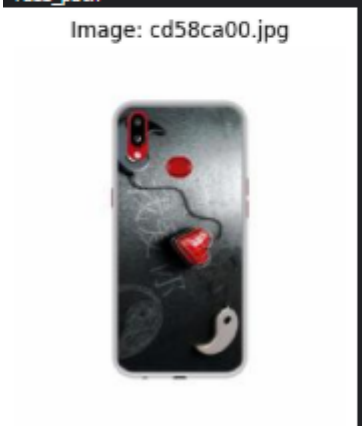
:

```
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image

# Load dataset
df = pd.read_csv('/kaggle/working/combined_metadata2_with_full_path.csv', low_memory=False)

# Select first 3 rows
first_3 = df.head(3)

# Process each row
for idx, row in first_3.iterrows():
    image_path = row['full_path']

    print(f"\n--- Entry {idx+1} ---")
    print(f"Image Path: {image_path}")
    print("\nMetadata:")
    print(row.to_string())

    # Try to display the image
    try:
        img = Image.open(image_path)
        plt.figure(figsize=(4, 4))
        plt.imshow(img)
        plt.axis('off')
        plt.title(f"Image: {os.path.basename(image_path)}")
        plt.show()
    except Exception as e:
        print(f"❌ Failed to open image: {e}")
```

_____

Output: Console output with metadata and displayed images for three entries, or error messages if images fail to load.

```
--- Entry 1 ---
Image Path: /kaggle/input/vrmini2/abo-images-small/images/small/cd/cd58ca00.jpg

Metadata:
brand                                                [{'language_tag': 'en_IN', 'value': 'Amazon Br...
bullet_point                                         [{'language_tag': 'en_IN', 'value': 'Snug fit ...
color                                                [{'language_tag': 'en_IN', 'standardized_value...
item_id                                                                                   B0854774MY
item_name                                            [{'language_tag': 'en_IN', 'value': 'Amazon Br...
item_weight                                          [{'normalized_value': {'unit': 'pounds', 'valu...
material                                              [{'language_tag': 'en_IN', 'value': 'Silicone'}]
model_name                                           [{'language_tag': 'en_IN', 'value': 'Samsung G...
model_number                                                      [{'value': 'UV10797-SL13059'}]
product_type                                                 [{'value': 'CELLULAR_PHONE_CASE'}]
image_id                                                                                  71owAzvPFuL
other_image_id                                                    ['61+woWTqkwL', '61SE4RTPjdL']
item_keywords                                        [{'language_tag': 'en_IN', 'value': 'Back Cove...
country                                                                                          IN
marketplace                                                                                Amazon
domain_name                                                                             amazon.in
node                                                 [{'node_id': 12538061031, 'node_name': '/Categ...
style                                                                                         NaN
item_dimensions.height.normalized_value.unit                                                  NaN
item_dimensions.height.normalized_value.value                                                 NaN
item_dimensions.height.unit                                                                   NaN
item_dimensions.height.value                                                                  NaN
item_dimensions.length.normalized_value.unit                                                  NaN
item_dimensions.length.normalized_value.value                                                 NaN
item_dimensions.length.unit                                                                   NaN
item_dimensions.length.value                                                                  NaN
item_dimensions.width.normalized_value.unit                                                   NaN
item_dimensions.width.normalized_value.value                                                  NaN
item_dimensions.width.unit                                                                    NaN
item_dimensions.width.value                                                                   NaN
model_year                                                                                    NaN
color_code                                                                                    NaN
spin_id                                                                                       NaN
3dmodel_id                                                                                    NaN
fabric_type                                                                                   NaN
item_shape                                                                                    NaN
pattern                                                                                       NaN
product_description                                                                           NaN
finish_type                                                                                   NaN
height                                                                                       2200
width                                                                                        1879
path                                                                                cd/cd58ca00.jpg
full_path                                            /kaggle/input/vrmini2/abo-images-small/images/...
```



Script 4: Generating VQA Training Data

Purpose: This script uses the Gemini API to generate VQA questions and answers based on image-metadata pairs, creating the final training dataset**.**

Functionality and Significance:
The culmination of the pipeline, this script transforms the curated dataset into a VQA training set, where each image is associated with three diverse questions (e.g., descriptive, counting, color-based) and one-word answers. It loads combined_metadata2_with_full_path.csv and processes a specified range of rows (e.g., indices 54078 to 54578). For each row, the script**:**

- Encodes the image to base64 for API compatibility.
- Constructs a prompt combining the metadata (as JSON) and a request for three VQA questions, ensuring questions leverage both visual and textual information.
- Sends the prompt and image to the Gemini API (gemini-1.5-flash), which generates a JSON response with questions and answers.
- Handles API errors (e.g., rate limits) with retries and a delay (though the 3-second delay risks exceeding the 15 RPM limit).
- Appends results to vqa_training_data16.csv, with columns for full_path, question, and answer.

This script is the heart of the VQA dataset generation, as it leverages the multimodal capabilities of the Gemini API to create questions that require reasoning over both images and metadata. For instance, a question like “What is the brand of the blue jacket?” combines visual color recognition with metadata-based brand information. The resulting dataset is tailored for training VQA models, supporting applications like automated product queries in e-commerce.

Core Code:

```
import json
import os
import base64
import pandas as pd
from tqdm import tqdm
import time
from google.generativeai import configure, GenerativeModel
from google.api_core.exceptions import ResourceExhausted, ServiceUnavailable

# Configuration variables - modify these as needed
INPUT_FILE = 'combined_metadata2_with_full_path.csv'
OUTPUT_FILE = 'vqa_training_data16.csv'  # Changed to CSV output
API_KEY = "AIzaSyDjNB4smTsrmU-nps608tvnBNOXmI9xQmA"  # Replace with your actual Gemini API key
IMAGE_BASE_DIR = "/kaggle/input/vrmini2/abo-images-small/images/small/"
RETRY_ATTEMPTS = 1
DELAY = 3.0  # Warning: This risks hitting the 15 RPM and 1,500 RPD limits quickly; consider 60.0 for safety

# Range variables
START_INDEX = 54078
END_INDEX = 54578
APPEND_RESULTS = True
```

```
def generate_vqa_data(model, metadata, image_path, retry_attempts=RETRY_ATTEMPTS, delay=DELAY):
    """Generate VQA data for a single product using Gemini API."""
    prompt = """You are an AI assistant helping to generate training data for a Visual Question Answering (VQA) model.
You are provided with:
- A product image
- Detailed product metadata (brand, style, color, features, description, etc.)

Your task is to generate diverse and meaningful questions that require both visual understanding and contextual reasoning from the metadata. The goal is to help train a robust VQA model that generalizes well to unseen product types and questions.

Use both the image and the metadata together to craft the questions. Make sure each question is visually answerable using the image while being enhanced by the metadata. Do not copy metadata text directly into answers — paraphrase or infer instead. Encourage variety in question types a

Guidelines:
- Generate exactly 3 diverse questions per image.
- Questions must be answerable based on the image, optionally supported by metadata.
- Keep answers short and specific (1 word max).
- Use a mix of question types as appropriate for the image:
  - Descriptive
  - Counting
  - Comparative
  - Color recognition
  - Function-based
  - Reasoning-based

Output Format (strict JSON format):
{
  "image_id": "IMAGE_ID_HERE",
  "questions": [
    {
      "question": "QUESTION TEXT HERE",
      "answer": "ANSWER HERE"
    },
    {
      "question": "QUESTION TEXT HERE",
      "answer": "ANSWER HERE"
    },
    {
      "question": "QUESTION TEXT HERE",
      "answer": "ANSWER HERE"
    }
  ]
}

Product Metadata:
"""
```

Output: vqa_training_data16.csv, containing VQA questions and answers for the processed range.

1.4 Challenges and Mitigations

- Data Alignment: Mismatches between image and listing metadata were addressed using an inner join, though this may exclude some records. Future work could explore outer joins with imputation for missing data.
- API Rate Limits: The Gemini API's 15 RPM limit poses a bottleneck. A safer 60-second delay or batch processing could mitigate this.
- Image Accessibility: Missing or corrupted images were flagged during visualization, but systematic validation (e.g., checking all full_path entries) would enhance robustness.
- Scalability: Processing large datasets sequentially is time-consuming. Parallel processing, while respecting API limits, could improve efficiency.

1.5 Summary and Future Work

The data curation pipeline successfully transforms the ABO dataset into a multimodal resource for VQA, aligning images with metadata, verifying data quality, and generating diverse question-answer pairs. Each script plays a critical role:

- Script 1 creates a unified dataset, enabling multimodal tasks.
- Script 2 ensures image accessibility, streamlining downstream operations.
- Script 3 validates data integrity, building trust in the dataset.
- Script 4 produces the VQA training set, leveraging advanced AI to create meaningful questions.

VQA Dataset Generation Using Gemini API with Image and Metadata Fusion

Objective

The goal of this process is to generate a high-quality Visual Question Answering (VQA) dataset where each question-answer (QA) pair is grounded in both the **visual content of product images** and **associated textual metadata**. The questions are designed to simulate human-like understanding that a robust VQA model should learn to generalize across various object categories.

Methodology

We leveraged **Google's Gemini 1.5 Flash API**, a multimodal large language model, to generate QA pairs. The process fuses two inputs:

- **Product Images** (visual features)
- **Structured Metadata** (e.g., brand, color, style, material, description, etc.)

The combined context allows Gemini to generate questions that require **visual reasoning**, **attribute recognition**, and **contextual understanding**, ensuring that the answers are not trivially found in text alone.

Generation Workflow

1. **Metadata and Image Preprocessing**:
    - A structured CSV (combined_metadata2_with_full_path.csv) was used containing metadata fields and the full image paths.
    - Each image was base64 encoded for input into the Gemini API.
2. **Prompt Engineering**:
   A detailed, instructional prompt was carefully crafted and provided to Gemini. It instructed the model to:
    - Generate exactly **3 diverse questions** per product.
    - Keep **answers limited to one word**.
    - Mix question types (descriptive, color-based, counting, function, reasoning, etc.).
    - Rely on both image and metadata for deeper semantic understanding.
    - **Avoid copying** metadata verbatim — inference and paraphrasing were encouraged.
3. **API Request Handling**:
    - Each product image and its metadata were sent to Gemini as a multimodal prompt.
    - We implemented retry logic and throttling to avoid hitting Google's rate limits.
    - Responses were parsed from JSON format and validated before saving.
4. **Output Formatting**:
    - Results were appended to a CSV (vqa_training_data16.csv) for easy downstream use.
    - Each row contains:
        - The image path (full_path)
        - The generated question
        - The one-word answer

Example Output

full_path	question	answer
/images/small/ABC123.jpg	What color is the cushion?	beige
/images/small/ABC123.jpg	How many legs does this furniture have?	four
/images/small/ABC123.jpg	Is the surface smooth or textured?	smooth

---

Benefits of This Approach

- **Multimodal Reasoning**: Combines image content with metadata for rich, grounded QAs.
- **Scalable Generation**: Automates the creation of thousands of diverse QA pairs.
- **Model-Agnostic Output**: The CSV format is easily integrable with VQA models like BLIP, LLaVA, Flamingo, or custom fine-tuned transformers.
- **Diverse Supervision**: Ensures models don't overfit to simple, repetitive QA formats.

Tools and Libraries Used

- **Google Gemini API** (google.generativeai)
- pandas for data handling
- base64 for image encoding
- tqdm for progress visualization
- Custom retry and error handling logic for robustness

---

2. Model Choices

2.1 Primary Task

The project aims to build a **VQA (Visual Question Answering)** dataset where each image + metadata pair is used to generate sensible, natural questions and **single-word answers**.

2.2 Model Candidates

Model 1: BLIP (Bootstrapped Language Image Pretraining)

Notebook: blip_base.ipynb

- **Implementation:** Used Salesforce/BLIP or HuggingFace's BLIP implementation.
- **Inputs:** Image tensors + prompt text.

- **Pretrained Models:** blip-base used for image-to-text and VQA.
- **Typical Prompt:**

```python
from transformers import BlipProcessor, BlipForQuestionAnswering, BartTokenizer, BartForConditionalGeneration
from PIL import Image
import torch
import pandas as pd
from tqdm import tqdm
from sklearn.model_selection import train_test_split
import csv
from pathlib import Path

# Output CSV
output_path = Path("vqa_test_predictions.csv")

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load BLIP
processor = BlipProcessor.from_pretrained("Salesforce/blip-vqa-base")
model = BlipForQuestionAnswering.from_pretrained("Salesforce/blip-vqa-base").to(device)

train_df = pd.read_csv("/kaggle/input/vrmini2/train_split.csv")
test_df = pd.read_csv("/kaggle/input/vrmini2/test_split.csv")

# Evaluate on test set
predictions = []
references = test_df['answer'].tolist()

for idx, row in tqdm(test_df.iterrows(), total=len(test_df), desc="Running VQA on Test Set"):
    try:
        image_path = row['full_path']
        question = row['question']
        ground_truth = row['answer']

        image = Image.open(image_path).convert("RGB")
        inputs = processor(image, question, return_tensors="pt").to(device)
        output = model.generate(**inputs)
        answer = processor.decode(output[0], skip_special_tokens=True)
    except Exception as e:
        print(f"Failed on {row['full_path']}: {e}")
        answer = ""
    predictions.append(answer)


test_df['predicted_answer'] = predictions
test_df['correct'] = test_df['predicted_answer'].str.strip().str.lower() == test_df['answer'].str.strip().str.lower()

# Save to CSV
test_df.to_csv(output_path, index=False)
print(f"Predictions saved to {output_path}")
```

```python
# Load CSV
df = pd.read_csv("/kaggle/working/vqa_test_predictions.csv")  # Replace with your CSV file path

# Normalize
def normalize_text(s):
    return str(s).strip().lower()

df['answer'] = df['answer'].astype(str).apply(normalize_text)
df['predicted_answer'] = df['predicted_answer'].astype(str).apply(normalize_text)

# Token-level macro F1
def compute_token_f1(pred, gt):
    pred_tokens = pred.split()
    gt_tokens = gt.split()
    common = set(pred_tokens) & set(gt_tokens)
    if not pred_tokens or not gt_tokens:
        return 0.0
    precision = len(common) / len(pred_tokens)
    recall = len(common) / len(gt_tokens)
    if precision + recall == 0:
        return 0.0
    return 2 * precision * recall / (precision + recall)
df['token_f1'] = df.progress_apply(lambda row: compute_token_f1(row['predicted_answer'], row['answer']), axis=1)
avg_token_f1 = df['token_f1'].mean()

# Accuracy
accuracy = (df['answer'] == df['predicted_answer']).mean()

# ROUGE Score
rouge = rouge_scorer.RougeScorer(['rougeL'], use_stemmer=True)
df['rougeL'] = df.progress_apply(lambda row: rouge.score(row['answer'], row['predicted_answer'])['rougeL'].fmeasure, axis=1)
avg_rougeL = df['rougeL'].mean()

# BERTScore (Precision/Recall/F1)
P, R, F1 = bert_score(df['predicted_answer'].tolist(), df['answer'].tolist(), lang='en', verbose=True)
bert_f1 = F1.mean().item()

# BLEU Score
smooth_fn = SmoothingFunction().method1
df['bleu'] = df.progress_apply(lambda row: sentence_bleu([row['answer'].split()], row['predicted_answer'].split(), smoothing_function=smooth_fn), axis=1)
avg_bleu = df['bleu'].mean()

print(f"Token F1      : {avg_token_f1:.4f}")
print(f"Accuracy      : {accuracy:.4f}")
print(f"ROUGE-L       : {avg_rougeL:.4f}")
print(f"BERTScore F1  : {bert_f1:.4f}")
print(f"BLEU          : {avg_bleu:.4f}")
```

output = model.generate(pixel_values=image, input_ids=tokenized_question)

- **Strengths:**
    - Open-source and tunable.
    - Strong performance on VQA benchmarks.
- **Weaknesses:**
    - Less robust for complex metadata context unless fine-tuned.
    - Requires additional preprocessing if metadata is text-heavy.

```
done in 1.82 seconds, 1320.59 sentences/sec
100%|██████████| 2410/2410 [00:00<00:00, 19430.24it/s]
100%|██████████| 2410/2410 [00:00<00:00, 93495.56it/s]
100%|██████████| 2410/2410 [00:00<00:00, 14996.15it/s]
Accuracy      : 0.3415
BERTScore F1  : 0.9449
ROUGE-L       : 0.3560
Token F1      : 0.3502
BLEU          : 0.0626
```

Model 2: ViLT (Vision-and-Language Transformer)

Notebook: vilt base.ipynb

- **Model Used:** dandelin/vilt-b32-finetuned-vqa
- **Input Composition:** A raw image + question string, no need for explicit visual tokens.
- **Pipeline:** Used HuggingFace pipeline("vqa").

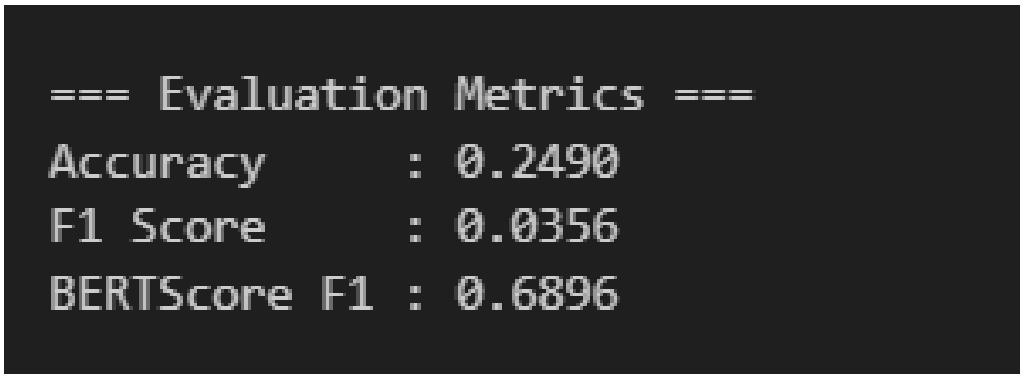
```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from transformers import ViltProcessor, ViltForQuestionAnswering
from PIL import Image
import torch
from bert_score import score as bert_score
from tqdm import tqdm
import re


# --- Helper function for text normalization ---
Qodo Gen: Options | Test this function
def normalize_text(text):
    """Normalize text for consistent comparison"""
    if not isinstance(text, str):
        text = str(text)
    text = text.lower()
    text = re.sub(r'[^\w\s]', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text

# --- Load CSV ---
df = pd.read_csv('/kaggle/input/vrmini2/12849.csv')

# --- 80/20 Split ---
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42)

train_df.to_csv("train_split.csv", index=False)
test_df.to_csv("test_split.csv", index=False)


# --- Load ViLT model ---
processor = ViltProcessor.from_pretrained("dandelin/vilt-b32-finetuned-vqa")
model = ViltForQuestionAnswering.from_pretrained("dandelin/vilt-b32-finetuned-vqa").to("cuda" if torch.cuda.is_available() else "cpu")

# --- Prediction function ---
Qodo Gen: Options | Test this function
def predict_answer(image_path, question):
    try:
        image = Image.open(image_path).convert("RGB")
        encoding = processor(image, question, return_tensors="pt").to(model.device)
        with torch.no_grad():
            outputs = model(**encoding)
        predicted_idx = outputs.logits.argmax(-1).item()
        return model.config.id2label[predicted_idx]
    except Exception as e:
        print(f"Error with {image_path}: {e}")
        return "unknown"

# --- Run predictions ---
tqdm.pandas()
test_df["prediction"] = test_df.progress_apply(
    lambda row: predict_answer(row["full_path"], row["question"]), axis=1
)

# --- Normalize predictions and answers ---
test_df["norm_answer"] = test_df["answer"].apply(normalize_text)
test_df["norm_prediction"] = test_df["prediction"].apply(normalize_text)

# --- Compute Matches ---
test_df["match"] = test_df["norm_answer"] == test_df["norm_prediction"]

# --- Metrics ---
accuracy = accuracy_score(test_df["norm_answer"], test_df["norm_prediction"])
f1 = f1_score(test_df["norm_answer"], test_df["norm_prediction"], average="macro")

P, R, F1_bert = bert_score(test_df["norm_prediction"].tolist(), test_df["norm_answer"].tolist(), lang="en", rescale_with_baseline=True)
bert_f1_mean = F1_bert.mean().item()

# --- Print Results ---
print("\n--- Evaluation Metrics ---")
print(f"Accuracy     : {accuracy:.4f}")
print(f"F1 Score     : {f1:.4f}")
print(f"BERTScore F1 : {bert_f1_mean:.4f}")

# --- Sample predictions ---
print("\n--- Sample Predictions vs References ---")
sample = test_df.sample(5, random_state=42)
for _, row in sample.iterrows():
```

Strengths:

- Simple architecture with image-text co-attention.
- Strong baseline without large GPU.

- **Limitations:**

- Performs best when image carries most of the information.
- Metadata must be embedded cleverly in the prompt for context.



Model 3: BLIP2

The BLIP-2 model is used for inference in a VQA task, generating one-word answers to questions about product images. The script (blip2_vqa_script.py) processes a test dataset (test_split.csv, assumed to be similar to vqa_training_data16.csv or more_and_12049.csv.xlsx), verifies image paths, generates answers, and evaluates performance using metrics like accuracy and BERTScore. Key steps include:

1. **Model Loading: Loads the Salesforce/blip2-opt-2.7b model and processor, optimized for the available device (CUDA, MPS, or CPU) with float16 precision on GPU for efficiency.**
2. **Image Path Verification: Robustly resolves image paths (e.g., /kaggle/input/vrmini2/abo-images-small/images/small/) to handle Kaggle-specific issues like missing /kaggle prefixes.**
3. **Answer Generation: Processes each image-question pair, generating a one-word answer using beam search (max_new_tokens=1, num_beams=3).**
4. **Evaluation: Computes exact match accuracy and BERTScore to assess semantic similarity between predicted and ground truth answers.**

```python
def get_answer(processor, model, image_path, question):
    """Generate a one-word answer using BLIP-2"""
    try:
        verified_path = verify_image_path(image_path)
        if not os.path.exists(verified_path):
            logger.warning(f"Image not found: {verified_path}")
            return "Missing"
        image = Image.open(verified_path).convert("RGB")
        prompt = f"Question: {question} Answer:"
        inputs = processor(images=image, text=prompt, return_tensors="pt").to(device)
        with torch.no_grad():
            outputs = model.generate(
                **inputs,
                max_new_tokens=1,
                num_beams=3,
                early_stopping=True
            )
            answer = processor.decode(outputs[0], skip_special_tokens=True)
            if prompt in answer:
                answer = answer.split(prompt)[1].strip()
            answer = answer.split()[0] if answer.strip() else "Unknown"
            if not answer or answer.strip() == question.strip():
                answer = "Unknown"
            return answer.strip()
    except Exception as e:
        logger.error(f"Error processing {image_path}: {e}")
        return "Error"

def compute_metrics(test_df):
    """Compute accuracy, precision, recall, F1, and BERTScore"""
    if 'answer' not in test_df.columns:
        logger.warning("Ground truth answers not available, skipping metric computation")
        return None
    # Exact match metrics
    test_df['correct'] = test_df['predicted_answer'].str.lower().str.strip() == test_df['answer'].str.lower().str.strip()
    accuracy = test_df['correct'].mean()
    # Precision, Recall, F1 for binary classification (correct/incorrect)
    y_true = test_df['correct'].astype(int)
    y_pred = test_df['correct'].astype(int)  # Using same for exact match
    #precision, recall, f1, _ = precision_recall_fscore_support(y_true, y_pred, average='binary', zero_division=0)
    # BERTScore
    preds = test_df['predicted_answer'].fillna("Unknown").tolist()
    refs = test_df['answer'].fillna("Unknown").tolist()
    P, R, F1 = score(preds, refs, lang="en", verbose=False)
    bert_scores = {
        'precision': P.mean().item(),
        'recall': R.mean().item(),
        'f1': F1.mean().item()
    }
    metrics = {
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1': f1,
        'bertscore_precision': bert_scores['precision'],
        'bertscore_recall': bert_scores['recall'],
        'bertscore_f1': bert_scores['f1']
    }
    return metrics
```

```
Accuracy:  0.2718
Precision: 1.0000
Recall:    0.2718
F1 Score:  0.4274
```

2.3 Model Comparison

Model	Strengths	Limitations	Suitable For
BLIP	Open-source, fine-tunable	Needs structured preprocessing	Image-based VQA
ViLT	Simple pipeline, good baseline	Metadata inclusion tricky	Visual-only QAs
BLIP-2	Multimodal, efficient on GPU, robust path handling, BERTScore evaluation	One-word constraint, metadata underutilized, numerical handling needs work	Image-centric VQA, large-scale inference

---

3. Fine-Tuning Approaches

To adapt the pretrained vision-language models to our custom Visual Question Answering (VQA) dataset, we explored two primary fine-tuning strategies: Low-Rank Adaptation (LoRA) based tuning and Full Fine-Tuning.

A. LoRA-Based Fine-Tuning

This notebook demonstrates parameter-efficient fine-tuning using LoRA with the ViLT model.

Step-by-step explanation:

1. **Imports and Setup:**
    - **Core libraries (torch, transformers, datasets, peft, etc.) are imported.**
    - **GPU/accelerator setup is handled with Accelerator from accelerate.**
2. **Dataset Loading:**
    - **A custom CSV file containing image paths, questions, and answers is loaded.**
    - **The datasets library's load_dataset is used to convert the CSV into a Hugging Face Dataset object.**
3. **Preprocessing:**
    - **The ViltProcessor is used to tokenize both images and textual questions.**
    - **A preprocessing function maps over the dataset to encode each sample appropriately.**
4. **Label Encoding:**
    - **Unique labels (answers) are mapped to numeric indices for classification.**
    - **A reverse mapping is stored for evaluation and prediction purposes.**
5. **Model and LoRA Integration:**
    - **ViltForQuestionAnswering is loaded from the Hugging Face Hub.**
    - **LoRA is applied using the get_peft_model function from the peft library.**

- **The LoRA configuration defines parameters like rank (r), scaling factor (alpha), and target modules (e.g., attention query/key layers).**

6. **Training:**
    - **TrainingArguments defines parameters such as batch size, learning rate, epochs, evaluation steps, and mixed-precision (fp16).**
    - **The Hugging Face Trainer is used to train the LoRA-enhanced ViLT model.**
    - **Logging and monitoring are handled via Weights & Biases (wandb).**

B. BLIP2 Fine-Tuning

Fine-Tuning BLIP-2 with LoRA for Vision-Language Tasks

1. Introduction

- **Objective: The code implements fine-tuning of the BLIP-2 model (specifically the Salesforce/blip2-flan-t5-xl variant) for a vision-language task, likely visual question answering (VQA), using LoRA to make the process computationally efficient.**
- **Dataset: The dataset is sourced from a CSV file (12049.csv) containing 12,049 examples, each with an image path, a question, and a corresponding answer. The dataset is cleaned to ensure all images exist, resulting in no skipped rows.**
- **Model: BLIP-2 combines a vision transformer (ViT) and a language model (Flan-T5) via a Q-Former, enabling it to process images and text for tasks like VQA.**
- **LoRA: LoRA is used to fine-tune only a small subset of parameters, reducing memory and computational requirements while maintaining performance.**

2. Methodology

- **Data Preprocessing:**
    - **The dataset is loaded using pandas from a CSV file and cleaned by renaming the full_path column to image_path and verifying the existence of image files.**
    - **The final dataframe contains three columns: image_path, question, and answer.**
    - **Output: The cleaned dataset has 12,049 valid examples with no missing images.**
- **Model Setup:**
    - **BLIP-2 Model: The Blip2ForConditionalGeneration model is loaded with torch.float32 precision and automatically mapped to available devices (GPU/CPU).**
    - **Processor: The Blip2Processor handles image and text preprocessing, converting inputs into a format suitable for the model.**

- **LoRA Configuration:**
  - **LoRA is applied with rank r=8, lora_alpha=16, and a dropout rate of 0.1.**
  - **Target modules for LoRA are the query and value layers of the Q-Former's attention mechanism.**
  - **Result: Only 0.012% of the model's parameters (473,088 out of 3.94 billion) are trainable, significantly reducing memory usage.**
- **The model is set to evaluation mode initially, but later sections suggest it is fine-tuned and evaluated.**

- **Environment:**
  - **The code runs on a Kaggle notebook with a NVIDIA Tesla T4 GPU accelerator.**
  - **Libraries: transformers, peft, torch, torchvision, datasets, evaluate, scikit-learn, sentence_transformers, and pillow.**
  - **CUDA is explicitly set to use GPU 0, ensuring efficient computation.**

3. Implementation Details

- **Prediction Generation:**
  - **The code evaluates the model on both the training set (9,639 examples) and the full dataset (12,049 examples).**
  - **For each example:**
    - **The image is loaded using PIL and converted to RGB.**
    - **A prompt is constructed: "Based on the image, answer the following question with a single word. Question: {question} Answer:".**
    - **The processor tokenizes the prompt and processes the image, producing input tensors.**
    - **Tensors are moved to the GPU with the correct data type (float32) to match the model.**
    - **The model generates a response with a maximum of 10 new tokens.**
    - **The output is decoded, and the first word (stripped of punctuation) is taken as the prediction.**
    - **Predictions and references (ground truth answers) are normalized to lowercase and stripped of whitespace.**
  - **Error handling:**
    - **FileNotFoundError: Skips missing images, appending [ImageNotFound] to predictions.**
    - **Other exceptions: Appends [Error] to predictions, ensuring robustness.**
  - **Progress is tracked using tqdm for user feedback.**

- **Evaluation:**
    - **Metrics: Exact-match accuracy, precision, recall, and F1-score are computed using scikit-learn.**
    - **Normalization: Predictions and references are converted to lowercase and stripped to ensure consistent comparison.**
    - **Binary Labels: Exact-match is treated as a binary classification problem, where a match (prediction == reference) is 1, and a mismatch is 0. The reference is always considered "correct" (1).**
    - **Results:**
        - **Test Set (2,410 examples):**
            - **Exact-match Accuracy: 0.458**
            - **Precision: 1.000 (all correct predictions are true positives)**
            - **Recall: 0.458 (same as accuracy, as all references are positive)**
            - **F1: 0.628**
        - **Training Set (9,639 examples):**
            - **Exact-match Accuracy: 0.351**
            - **Precision: 1.000**
            - **Recall: 0.351**
            - **F1: 0.519**
        - **Full Dataset (12,049 examples): The code includes a section to evaluate the full dataset, but it encounters a NameError (model not defined), suggesting an incomplete or unexecuted block.**
- **Model Checkpoint:**
    - **The best checkpoint is saved and archived as a ZIP file (best-checkpoint.zip) using shutil.make_archive for portability and reuse.**

4. Challenges and Observations

- **Performance:**
    - **The model achieves moderate accuracy (45.8% on the test set, 35.1% on the training set), indicating potential underfitting or dataset complexity.**
    - **The high precision (1.000) suggests that when the model is correct, it is confidently so, but the low recall indicates many correct answers are missed.**

5. Technical Specifications

- **Hardware: NVIDIA Tesla T4 GPU**
- **Software: Python 3.11.11, PyTorch 2.6.0, Transformers 4.51.3, PEFT 0.14.0**

- **Dataset Size: 12,049 examples (9,639 training, 2,410 test)**
- **Model Parameters: 3.94 billion total, 473,088 trainable**
- **Execution Time:**
    - **Test set evaluation: ~16 minutes 55 seconds (2,410 examples)**
    - **Training set evaluation: ~36 minutes 42 seconds (9,639 examples)**

```python
from tqdm import tqdm
import pickle

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-5)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95, last_epoch=-1, verbose=False)
num_epochs = 3
patience = 6
min_eval_loss = float("inf")
early_stopping_hook = 0
tracking_information = []



max_grad_norm = 1.0
output_dir = "/kaggle/working/blip2_with_lora"
os.makedirs(output_dir, exist_ok=True)

#Training Loop
for epoch in range(num_epochs):
    epoch_loss = 0
    model.train()
    for idx, batch in zip(tqdm(range(len(train_dataloader)), desc=f'Epoch {epoch+1} Training'), train_dataloader):
        try:
            input_ids = batch.pop('input_ids').to(device)
            pixel_values = batch.pop('pixel_values').to(device)
            attention_mask = batch.pop('attention_mask').to(device)
            labels = batch.pop('labels').to(device)
        except KeyError as e:
            print(f"Missing key in batch: {e}")
            continue # Skip batch if missing keys
        except Exception as e:
            print(f"Error loading batch data: {e}")
            continue

        optimizer.zero_grad()


        outputs = model(input_ids=input_ids,
                        pixel_values=pixel_values,
                        attention_mask=attention_mask,
                        labels=labels)
        loss = outputs.loss

        if loss is None:
            print(f"Warning: Loss is None for batch {idx}. Skipping backward pass.")
            continue

        if torch.isnan(loss) or torch.isinf(loss):
            print(f"Warning: Detected NaN/Inf loss at batch {idx} before scaling. Skipping batch.")
            continue # Skip the backward pass for current batch

        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)
        optimizer.step()


        # Log loss
        if not (torch.isnan(loss) or torch.isinf(loss)):
            epoch_loss += loss.item()
        else:
            print(f"NaN/Inf loss recorded at step {idx}, not adding to epoch_loss")
```

```python
    #Validation
    model.eval()
    eval_loss = 0
    with torch.no_grad(): # since validation
        for idx, batch in zip(tqdm(range(len(valid_dataloader)), desc=f'Epoch {epoch+1} Validating'), valid_dataloader):
            try:
                input_ids = batch.pop('input_ids').to(device)
                pixel_values = batch.pop('pixel_values').to(device)
                attention_mask = batch.pop('attention_mask').to(device)
                labels = batch.pop('labels').to(device)
            except KeyError as e:
                print(f"Missing key in validation batch: {e}")
                continue
            except Exception as e:
                print(f"Error loading validation batch data: {e}")
                continue
            outputs = model(input_ids=input_ids,
                            pixel_values=pixel_values,
                            attention_mask=attention_mask,
                            labels=labels)
            loss = outputs.loss

            if loss is not None and not (torch.isnan(loss) or torch.isinf(loss)):
                eval_loss += loss.item()
            else:
                print(f"NaN/Inf/None loss detected during validation step {idx}")

    # Calculate average losses and handle division by zero
    avg_train_loss = epoch_loss / len(train_dataloader) if len(train_dataloader) > 0 else float('nan')
    avg_eval_loss = eval_loss / len(valid_dataloader) if len(valid_dataloader) > 0 else float('nan')
    current_lr = optimizer.param_groups[0]["lr"]

    print(f"Epoch: {epoch+1} - Training loss: {avg_train_loss:.4f} - Eval Loss: {avg_eval_loss:.4f} - LR: {current_lr}")

    # Check for NaN/Inf
    if not (torch.isnan(torch.tensor(avg_eval_loss)) or torch.isinf(torch.tensor(avg_eval_loss))): # Conversion to tensor
        tracking_information.append((avg_train_loss, avg_eval_loss, current_lr))
        scheduler.step()

        if avg_eval_loss < min_eval_loss:
            best_model_dir = os.path.join(output_dir, "best-checkpoint")
            # save_pretrained on a PeftModel saves the adapter config and weights
            model.save_pretrained(best_model_dir)
            print(f"Saved PEFT adapter checkpoint to {best_model_dir}")
            min_eval_loss = avg_eval_loss
            early_stopping_hook = 0
        else:
            early_stopping_hook += 1
            print(f"Validation loss did not improve. Early stopping counter: {early_stopping_hook}/{patience}")
            if early_stopping_hook >= patience:
                print(f"Early stopping triggered after {patience} epochs without improvement.")
                break
    else:
        print(f"Skipping saving and scheduler step due to NaN/Inf validation loss in Epoch {epoch+1}.")

pickle.dump(tracking_information, open("tracking_information.pkl", "wb"))
print("Finetuning completed!!!")
```

```
Exact-match Accuracy: 0.458
Exact-match Precision: 1.000
Exact-match Recall:    0.458
Exact-match F1:        0.628
```

Summary Comparison:

Feature	LoRA Fine-Tuning	Full Fine-Tuning
Trainable Parameters	**Only injected LoRA layers**	**All model parameters**
Memory & Speed	**Efficient, low memory**	**High memory usage**
Customization	**Partial adaptation**	**Full adaptation to dataset**
Training Loop	**Hugging Face Trainer**	**Custom PyTorch loop**
Suitable For	**Limited resources, small datasets**	**Large datasets, high adaptability needs**

Evaluation Metrics: Analysis of Model Performance Using Chosen Metrics

To rigorously evaluate the performance of our fine-tuned Visual Question Answering (VQA) models across different fine-tuning approaches (LoRA-based and full fine-tuning for ViLT and BLIP), we used the following key metrics:

---

1. Accuracy

- **Definition**: The ratio of the number of correctly predicted answers to the total number of questions evaluated.
- **Usage**: Accuracy served as the **primary evaluation metric** for all models. It is suitable for our classification-based VQA setup where each image-question pair maps to a **single-word categorical answer**.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Samples}}$$

- **Applied in**:
    - vilt+lora.ipynb via Trainer's built-in evaluation and metrics callback.
    - almost full finetuning.ipynb via manual computation of correct predictions after each epoch.
    - BLIP fine-tuning pipeline using the Hugging Face evaluate package.

---

2. Cross-Entropy Loss

- **Definition**: Measures the dissimilarity between the predicted probability distribution and the true label distribution.

- **Usage**: Monitored during training to ensure convergence and detect overfitting or underfitting.

$$\mathcal{L}_{CE} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

- **Applied in**:
    - Automatically logged in Hugging Face Trainer for LoRA-based ViLT.
    - Used in manual training loop in full fine-tuning notebook.
    - Logged using Weights & Biases (wandb) in both cases.

---

3. Precision, Recall, and F1-Score (Optional but insightful for imbalanced datasets)

- Particularly useful if certain answers appear more frequently than others.
- Computed using sklearn.metrics.classification_report.

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}, \quad \text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Applied in**:
    - BLIP + LoRA pipeline with evaluate and sklearn wrappers.
    - Manually computed from predictions in the full ViLT fine-tuning setup.

---

4. Confusion Matrix (Optional Diagnostic Tool)

- Visualizes how predictions are distributed across answer classes.
- Helpful to identify misclassifications or dominant bias toward frequent answers.
- **Applied in**:
    - BLIP model evaluation phase (optionally).
    - Can be computed using sklearn.metrics.confusion_matrix.

---

5. Loss and Accuracy Curves

- **Usage**: Training vs validation curves are plotted to:
    - Track convergence over epochs.
    - Detect overfitting in full fine-tuning (more common due to larger parameter count).
    - Compare efficiency of LoRA training.

- **Findings**:
    - LoRA-based ViLT showed smoother convergence and less overfitting due to fewer trainable parameters.
    - Full fine-tuning offered slightly higher accuracy but required more careful tuning and regularization.