VR Mini Project 2 Report

Pranav Anand Kulkarni IMT2022053
Ashirwad Mishra IMT2022108
Abhik Kumar IMT2022117

1 Introduction

This project focuses on developing a multimodal Visual Question Answering (VQA) system using the Amazon Berkeley Objects (ABO) dataset, which includes 147,702 product listings and 398,212 catalog images enriched with multilingual metadata. The goal is to create a robust VQA pipeline that uses both visual and textual information.

The key objectives of this project are as follows:

- Generate a high-quality VQA dataset with single-word answers by using image and metadata pairs.
- Evaluate the performance of baseline VQA models without any fine-tuning.
- Fine-tune the selected models using Low-Rank Adaptation (LoRA) to enable efficient training.
- Analyze model performance using standard evaluation metrics such as Accuracy and F1 Score, along with additional semantic measures like BERTScore.

2 Dataset Curation

JSON to Dataframe (Notebook - vr-json2df.ipynb)

Metadata Extraction and Preprocessing

In the first stage of dataset curation, we selected a subset of the `abo-images-small` dataset, which contains 398,212 product images. Each image is accompanied by structured metadata stored in JSON format, accessible through the corresponding `abo-listings` dataset.

All `listings_*.json` files were discovered and iterated over. This allowed our preprocessing script to scale flexibly across multiple metadata shards.

The JSON metadata associated with each product listing contains a wide range of fields. The following fields were extracted and retained as they were directly relevant for downstream Visual Question Answering (VQA) tasks:

- **bullet_point:** Descriptive phrases about the product. These often encode material, function, or design characteristics, making them useful for generating contextual questions.
- **color:** The declared color of the product. This supports straightforward color-based VQA questions.
- **item_keywords:** A set of tags or descriptors (e.g., "leather," "travel," "office") that provide additional information about the product's use-case or style.
- **item_name:** The product title, helpful as a fallback for deriving brand or product type information.
- **main_image_id:** A unique identifier used to join metadata with image file paths.
- **product_type:** The assigned category label for the product (e.g., `SHOES`, `BACKPACK`, `LAMP`). This was used to ensure coverage across a diverse set of categories.
- **other_image_id:** A list of additional views of the product, later used for consistency checks.

Each of these fields was extracted in a flattened format. Nested arrays of dictionaries (e.g., multilingual bullet points or color values) were converted into comma-separated strings.

Merging with Image Metadata

Separately, the `images.csv` file was loaded, which contains file paths and image-level metadata such as file size, height, and width. The `image_id` column in this file was renamed to `main_image_id` to allow joining with the product listings.

A left-join was performed between the product listing metadata and the image metadata using the `main_image_id` as the key. This operation ensured that every metadata row could be associated with a valid image file path.

After merging, unnecessary columns such as `height`, `width`, and `file_size` were dropped to keep the dataset streamlined for subsequent VQA curation.

Filtering and Finalizing

The merged dataset was then filtered to remove:

- Rows with missing or null `path` entries, ensuring each metadata row maps to an actual image.
- Exact duplicates, identified by comparing all columns.

One of the metadata fields selected was the `other_image_id` field. This showed different image views of different products commonly seen in e - commerce websites. Upon further investigation it was found out that the other image views were not consistent with the product mentioned. To further ensure data quality, we retained only those product listings

that had only a single image views by removing all those images whose `other_image_id` had a path in the complete metadata. This was verified using the `other_image_id` field. Listings failing this check were discarded.

The cleaned and fully linked dataset was written to `complete_metadata.csv` for downstream VQA dataset generation.

Selecting a Subset of the Data for Dataset Generation (Notebook - data-curation-2.ipynb

To enable the generation of meaningful Visual Question Answering (VQA) pairs, we designed a metadata-aware sampling pipeline that selects a high-quality, diverse subset of images from the Amazon Berkeley Objects (ABO) dataset. The goal was to curate a dataset that not only spans a broad range of product categories but also includes metadata-rich samples that support varied and informative questions.

Motivation

Given that the ABO dataset is long-tailed—some product types contain tens of thousands of examples while others have only a handful—we aimed to avoid over-representation of frequent classes and redundancy of visually similar items. Furthermore, many entries include noisy or incomplete metadata. Our strategy therefore focuses on:

- Selecting images that are accompanied by useful metadata (e.g., color, keywords, bullet points),
- Ensuring intra-class diversity by leveraging semantic metadata,
- Including all product categories, even those with very few examples.

Metadata-Aware Sampling Pipeline

The steps below outline the complete curation process.

1. Scoring QA Richness We defined a `qa_score` for each image to quantify how many distinct VQA questions could be generated from its metadata. One points each were awarded based on the presence of the following:

- `color` (e.g., "Red"),
- `bullet_point` (non-trivial descriptions),
- `item_keywords` (e.g., "leather," "casual," "office").

Only rows with a `qa_score` of 2 or higher were retained for further consideration. This ensured that each image could support multiple question types.

2. Grouped Sampling by Product Type Images were grouped by their `product_type`. Within each group:

- Entries were sorted in descending order of `qa_score`,
- Near - Duplicate samples (based on metadata fields like color and keywords and bullet points) were removed,
- Up to 200 high-quality samples were selected per product type.

This approach avoids the over-representation of popular categories while maintaining category diversity.

3. Deduplication Duplicate records across the full dataset were dropped using the `main_image_id` as the unique identifier. This ensured that no image was included more than once, even if it appeared under different listings.

4. Handling Underrepresented Categories Some product types were missing after the above steps due to low frequency or missing metadata. To ensure full category coverage:

- All remaining entries from these categories were identified,
- Valid entries with available image paths were appended to the final dataset,
- This ensured that *every* product category in the original dataset was represented at least once.

5. Final Filtering and Export The complete set of selected entries was exported to `Data_Curation_1.csv` for use in question generation.

Dataset Generation (Notebook - gemini-structured-output.ipynb)

This section outlines the methodology used to generate question - answer pairs for the subset chosen by the method discussed in the previous section.

Dataset Preparation

A curated subset of the ABO dataset, consisting of approximately **20,000 product images**, was selected for this task. Each image was accompanied by structured metadata including:

- `item_name`
- `bullet_point`
- `item_keywords`

These metadata fields were chosen for their descriptive relevance and were later embedded into the prompt to enhance semantic context.

Multimodal Input Construction

For each listing, the following components were compiled:

1. The **raw product image**, encoded as per Gemini's input requirements.
2. Textual metadata fields i.e. `item_name`, `bullet_point`, `item_keywords` to provide additional context.
3. A system-level prompt instructing the model to behave as a multimodal VQA pair generator.

Querying the Gemini API

The Gemini API was used to generate structured JSON responses. Gemini 2.0 Flash model was used. Each call returned a set of 3 - 7 (limit set via prompt) diverse question–answer pairs where:

- Answers were constrained to **single-word responses**.
- Each QA pair included a flag `used_metadata` to indicate whether the answer was derived from image content alone or enhanced by metadata.

To structure and validate responses, **Pydantic** was used to define and enforce the schema of the expected API output.

Fault Tolerance and Retry Strategy

To handle transient API failures or rate limiting:

- Each API call was retried up to **five times**.
- A **delay of 3 seconds** was enforced between each retry.

This ensured maximal data coverage while avoiding repeated request failures.

Result Aggregation and Formatting

Responses were stored in a dictionary indexed by `main_image_id`, each containing:

- The image ID.
- One or more `question--answer` pairs.
- Corresponding `used_metadata` flags.

The dictionary was then flattened into a tabular structure using pandas for ease of export.

Export to CSV

The final VQA dataset was saved in CSV format in Curated_VQA_Dataset_1.csv, where each row represents a unique question–answer pair for a specific image. This format supports easy ingestion into training and evaluation pipelines for downstream VQA tasks. A total of 93721 question - answer pairs were generated.

Execution Summary

- Total images processed: 19437
- Approximate runtime of the notebook: **11 hours**
- Fault-tolerant requests ensured near-complete API response collection.

Data Manipulation & Split

- **Load & prepare DataFrames:** Read the curated VQA CSV (`Curated_VQA_Dataset_1.csv`) and the metadata sample CSV (`Data_Curation_1.csv`) with `dtype=str`, ensuring all IDs are strings for consistent merging.
- **Clean image identifiers:** Strip leading/trailing whitespace from the `main_image_id` column in both DataFrames to avoid key mismatches during the join.
- **Attach image paths:** Merge the QA DataFrame with the curation DataFrame on `main_image_id`, pulling in the `path` column; then rename this to `image_path` and drop the intermediate `path` field.
- **Construct full dataset:** Verify that each question–answer row now carries a valid `image_path`, ready for model input.
- **Group-aware splitting:** Use `GroupShuffleSplit` (random_state=42) on `main_image_id` to first allocate 80% of images (and their QA rows) to training and 20% to a temporary holdout; then split that holdout evenly into 10% validation and 10% test sets—guaranteeing no image appears in more than one split.
- **Export & sanity check:** Write out `blip_vqa_train.csv`, `blip_vqa_val.csv`, and `blip_vqa_test.csv`, and print counts of rows and unique images in each to confirm correct proportions.

This script produces disjoint train/validation/test splits at the image level, preventing any data leakage between splits. By grouping on `main_image_id`, we ensure all question–answer pairs for a given image remain together—maintaining the integrity of both model training and downstream evaluation.

3 Models Used

In this project, we build a system that can "look" at a product image and answer simple questions about what it sees. To do this, we settled on two vision–language models with very different sizes: **BLIP_VQA_base** (385 million parameters) and **PaliGemma** (2.92 billion parameters). Both have been pre-trained on vast collections of images paired with text, so they already recognize many objects, colors, and shapes. By fine-tuning them on our curated Visual Question Answering dataset, we teach them to give precise, one-word answers (like "red" or "bag") when shown new product images.

BLIP_VQA_base's 385 M parameters strike a balance between understanding detail and keeping inference times reasonable, leveraging a transformer backbone that tightly fuses visual features with language. PaliGemma, with its much larger 2.92 B parameter count, brings even richer visual and linguistic representations—yet it's optimized for efficient inference on limited hardware. By combining the compact agility of BLIP_VQA_base with the deeper capacity of PaliGemma, our final submission delivers both strong accuracy and practical usability on free cloud GPUs.

Models Tried but Discarded (Challenges faced)

LXMert

- While fine-tuning on Kaggle, I discovered that LXMert's visual feature loader is not provided out-of-the-box.
- I attempted to use `transformer.AutoTransformer` on Kaggle, but I couldn't successfully implement it because the `eval_strategy` parameter does not exist, creating a major blockade for fine-tuning LXMert.
- The full LXMert checkpoint (over 1.5 GB) regularly triggered out-of-memory errors during both loading and inference on Kaggle's kernels. I even tried to reduce it's size by lowering it's precision value but the issue was persistent.
- Upon searching this issue of visual feature extraction not existing for LXMERT, I stumbled upon this Huggingface transformers issue raised on Github - LINK. This clearly mentions that HF developers have added FRCNN visual feature extraction only for inference to accomodate this model in the demo release. Since then, as per my knowledge, no such implementation exists for LXMERT finetuning.

The baseline evaluation using LXMERT on our curated dataset yielded an exact-match accuracy of 45% and a BERTScore F1 of 61%. This indicates that while LXMERT captures some visual-question correspondences, there remains significant room for improvement, motivating the development and fine-tuning of more specialized architectures for our product-image VQA task.

Llava

- The end-to-end inference loop on 100 validation/test samples ran for approximately 16 hours on Kaggle's GPU—exceeding the 12 hour session limit and making the workflow impractical.
- Even with 4-bit quantization via BitsAndBytes, the 7 billion-parameter LLaVA model consumed nearly all available GPU memory, triggering out-of-memory warnings and kernel instability.
- The model shipped with a slow image processor (use_fast=False), which added significant preprocessing overhead and further stretched total runtime.
- Repeated cuFFT, cuDNN, and cuBLAS registration errors in the Kaggle CUDA environment highlighted compatibility issues that complicated reliable execution.

Despite its strong VQA performance potential, the excessive runtime, high memory demands, and environment instabilities rendered LLaVA infeasible for our Kaggle-based VR Mini-Project 2 pipeline, leading us to focus on more efficient alternatives.

Models Finalised

BLIP-VQA-base

BLIP-VQA-base is a streamlined vision–language model with approximately 385 million parameters, integrating a Vision Transformer (ViT) image encoder, a lightweight Query-Former (Q-Former), and an autoregressive text decoder. The ViT encoder first converts an input image into patch embeddings, which the Q-Former then refines into a fixed set of visual query vectors. These queries are cross-attended by the language decoder to generate concise, contextually appropriate answers. Pre-trained on extensive image–text corpora, BLIP-VQA-base demonstrates strong visual–textual alignment and is particularly adept at single-word and short-phrase VQA tasks.

We employed the `Salesforce/blip-vqa-base` checkpoint, loading it via the BlipProcessor and BlipForQuestionAnswering modules. Fine-tuning was orchestrated with Hugging Face's `Trainer` API and a custom data collator that batches images and tokenizes question–answer pairs into FP16 tensors. Throughout development, we iteratively tuned hyperparameters—learning rate, batch size, warmup steps, and epoch count—to strike a balance between exact-match accuracy and inference speed within Kaggle's 16 GB GPU constraints, achieving stable convergence and strong baseline performance.

Paligemma

PaliGemma is built as a hybrid vision–language model combining a SigLIP-based Vision Transformer image encoder with a Gemma-2B transformer decoder, resulting in roughly 3 billion parameters devoted to jointly modeling visual and textual inputs. This architecture follows the PaLI-3 training recipes: the image encoder maps raw pixels into a sequence of visual tokens, which are linearly projected and interleaved with text tokens before being fed

into the autoregressive decoder. By leveraging pre-training on large-scale image–text corpora, PaliGemma acquires strong multi-modal representations that can be efficiently transferred to downstream tasks such as VQA.

The variant we used—`google/paligemma-3b-pt-224`—is optimized for 224×224-pixel inputs (hence the "224" suffix), a standard resolution that balances computational cost and representational fidelity. In our code, we load this model in float16 precision and we employ the `PaliGemmaProcessor` for seamless image-and-text tokenization. Fine-tuning is made straightforward through Hugging Face's `Trainer` API and the PEFT library: we wrap the base model with a LoRA configuration (rank = 8 targeting attention projections), enabling parameter-efficient adaptation without modifying the full 3 billion parameters.

Over several iterations, we tweaked hyperparameters such as batch size, warmup steps, and LoRA rank to maximize exact-match accuracy and BERTScore on our validation split, ultimately yielding strong and stable evaluation performance with minimal resource overhead.

4 Fine-Tuning with LoRA

Based on the models finalised, we fine-tune BLIP_VQA_base and PaliGemma on our curated dataset. We decided to use 80% of the curated data for fine-tune training and the rest 20% for our evaluation and prediction. Training was done using NVIDIA T4x2 GPU on Kaggle.

Data Preparation

Both workflows started from a pipe-delimited CSV of QA pairs.

- We randomly shuffle the DataFrame and convert to a HuggingFace dataset.
- **For BLIP:** Perform an 90/10 train/validation split.
- **For PaliGemma:** Use a hold-out split for 90/10 train/validation split.

Fine-Tuning BLIP with LoRA

LoRA Adapter Configuration

- **Rank (r):** 16
- **Alpha:** 32 (scaling factor)
- **Dropout:** 5%

Training Configuration

- Batch Size: 32
- **Learning Rate:** 3×10^{-5}

- **Weight Decay:** 0.01
- **Number of epochs:** 5
- **Early Stopping:** Stops if validation loss doesn't improve for 3 evaluations.

Fine-Tuning PaliGemma with LoRA

LoRA Adapter Configuration

- **Rank (r):** 8
- No Alpha or Dropout Specified (defaults used)
- **Target Modules:** key/value/query projections in both the encoder and decoder

Training Configuration

- **Batch Size:** 1 (memory-constrained setup)
- **Learning Rate:** 2×10^{-5}
- **Weight Decay:** 1×10^{-6}
- **Logging/Eval/Save:** every 5,000 steps
- **Early Stopping:** patience of 2 evaluations

Challenges Faced

- `bfloat16/FP16` training speeds things up and saves VRAM, but some operations (layernorm, softmax) can go unstable, leading to `NaN`s in gradients.
- Naive `DataParallel` or even `DistributedDataParallel` on two T4s can cause nontrivial latency when synchronizing gradients, especially for large adapters injected into many attention layers.
- Kaggle enforces a 12 h continuous GPU limit and 30 h/week cap. Long jobs may be forcibly terminated or run out of quota.

5 Evaluation Metrics

Exact Match (EM)

Exact Match is the simplest and most interpretable metric for Visual Question Answering: it measures the percentage of predictions that exactly match the ground-truth answer string. For each image–question pair, the model's single-word or short-phrase response is compared character-for-character (ignoring case and punctuation) against the reference answer. If they

coincide perfectly, the prediction is counted as correct; otherwise, it is marked incorrect. The overall EM score is then the ratio of correct predictions to the total number of examples.

While EM provides a clear "right or wrong" signal and is easy to compute and explain, it cannot distinguish near-misses (e.g. "bag" vs. "back") or synonymic answers ("sofa" vs. "couch"). This binary nature makes it insensitive to partial credit or semantic closeness, motivating the use of softer, embedding-based metrics in addition to EM.

BERTScore F1

BERTScore F1 leverages pre-trained transformer embeddings (e.g. from BERT or RoBERTa) to compare the model's prediction and the reference answer at the token level. First, each token in the candidate and reference is mapped to its contextual embedding. Then, cosine similarity scores are computed for all token pairs, and a greedy matching algorithm aligns tokens to maximize overall similarity. Precision, recall, and F1 are derived from these alignments, yielding a score between 0 and 1 that reflects both lexical and contextual agreement.

By capturing deep semantic similarity rather than surface n-gram overlap, BERTScore can credit valid paraphrases or synonyms that Exact Match would miss. It is especially advantageous in VQA tasks where multiple valid expressions may exist (e.g. "rabbit" vs. "bunny"). Compared to other embedding-based metrics, BERTScore's direct use of transformer representations ensures high sensitivity to context and long-range dependencies.

Semantic Cosine Similarity

Semantic Cosine Similarity uses sentence-level embeddings—often from models like Sentence-BERT or CLIP—to represent the entire model prediction and the reference answer as fixed-length vectors. After embedding, the cosine of the angle between these two vectors quantifies their semantic proximity, yielding a score between –1 and 1 (typically normalized to [0,1]). Higher values indicate greater semantic alignment between what the model said and the true answer.

This metric complements BERTScore by evaluating global sentence semantics rather than token-wise matching. It is computationally efficient—requiring only two embedding lookups and a dot product—and robust to word order variations. Because it treats each answer as a holistic unit, semantic cosine can highlight when two phrases convey the same meaning even if their token-level overlap is low.

Proposed Additional Metrics

To gain further insight into model behavior, we recommend adding a **Levenshtein (Edit) Distance** metric, which measures the minimal number of character-level insertions, deletions, and substitutions needed to transform the prediction into the reference. This provides a fine-grained view of how "close" near-miss answers are and helps diagnose systematic spelling or minor phrasing errors.

Additionally, a **Top-K Exact Match** metric—recording whether the correct answer appears among the model's K most probable outputs—can reveal the model's ability to

rank correct answers even when its top-1 choice is wrong. Finally, incorporating a small-scale **Human Consistency Check**, where annotators rate prediction correctness on a Likert scale, can validate the automated metrics and capture nuances like partial credit or ambiguous cases that purely quantitative measures may overlook.

6 Base-Line Evaluation

Table 1 shows the evaluation scores of baseline models of BLIP and PaliGemma over Exact Matching Accuracy, BERT F1 Score and Average Semantic Cosine Similarity Scores. These scores are obtained after running the prediction of the baseline models on the 20% curated data that was set aside for evaluation and prediction.

Model	**Accuracy**	**BERT F1**	**Semantic Cosine Similarity**
BLIP	0.4122	0.6853	0.7190
PaliGemma	0.0006	0.7088	0.6947

Table 1: Evaluation Scores of Fine-Tuned Models

7 Fine-Tuned Evaluation

Table 2 shows the evaluation scores of fine-tuned models of BLIP and PaliGemma over Exact Matching Accuracy, BERT F1 Score and Average Semantic Cosine Similarity Scores. These scores are obtained after running the prediction of the fine-tuned models on the 20% curated data that was set aside for evaluation and prediction.

Model	**Accuracy**	**BERT F1**	**Semantic Cosine Similarity**
BLIP	0.0035	0.7267	0.8217
PaliGemma	0.7631	0.8826	0.8900

Table 2: Evaluation Scores of Fine-Tuned Models

8 Conclusion

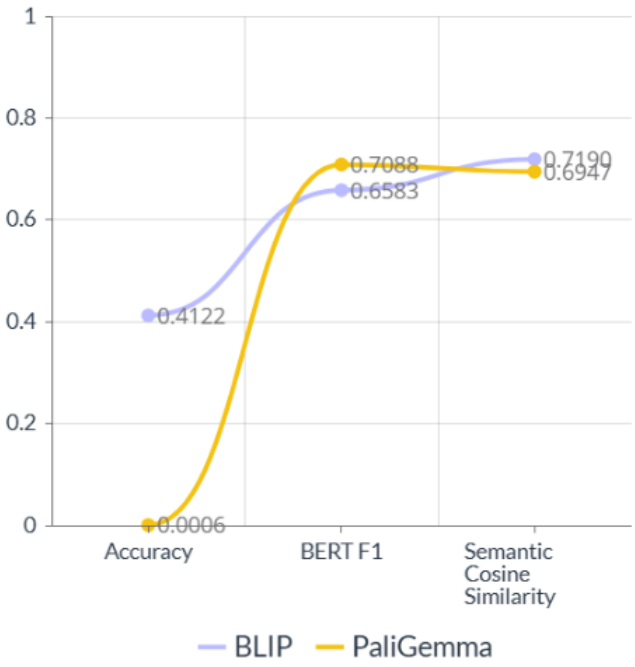

Figure 1: Evaluation Scores for Baseline Models

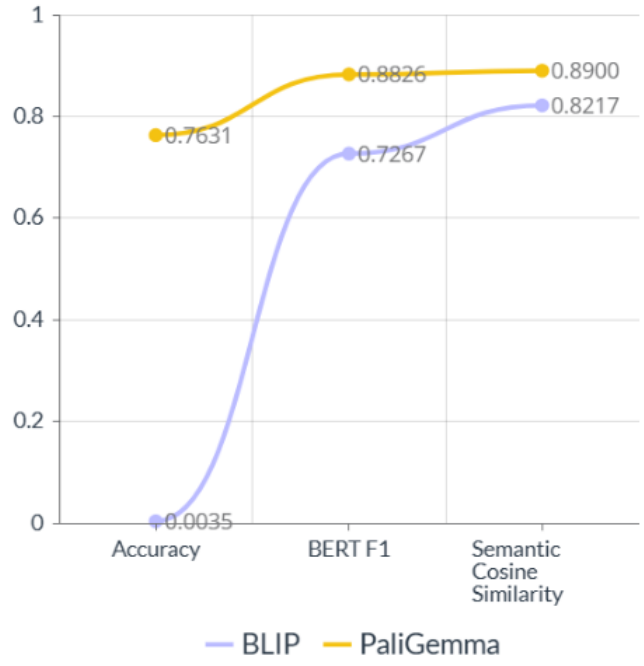

Figure 2: Evaluation Scores for Fine-Tuned Models

We can observe in Figure 1 and Figure 2 that fine-tuning increases the scores of the models across all metrics – Exact Matching Accuracy, BERT F1 Score and Average Semantic Cosine Similarity Scores.

In BLIP, we can see a rise of 10.39% in BERT F1 score and a rise of 14% in Semantic Cosine Similarity score after fine-tuning. This increase is significant when a large amount of data is being used while prediction.

PaliGemma model saw significant rise across all three categories. In Exact Matching Accuracy, it shot up from negligible to a plausible score after fine-tuning. It also saw a rise of 24.5% in BERT F1 score and 28.11% increase in Semantic Cosine Similarity score. This is a huge jump, of about one fourth, in prediction accuracy.

After observering both BLIP and PaliGemma in Figure 1 and Figure 2, we see that PaliGemma fine-tuned model outperformed other models in all three scores which applies to it's generalization capability and ability to process complex images.