Multimodal Visual Question Answering

with Amazon Berkeley Objects (ABO) Dataset

Aryaman Pathak IMT2022513
Rutul Patel IMT2022021
Shreyas Biradar IMT2022529

Project Overview

Using the extensive **Amazon Berkeley Objects (ABO)** dataset, this project focuses on creating a **Multimodal Visual Question Answering (VQA)** system. We aim to utilize the **Low-Rank Adaptation (LoRA)** method to generate a multiple-choice VQA dataset from ABO, establish and evaluate robust baseline models, and significantly enhance their performance through fine-tuning. The ultimate goal is to build a high-performing VQA model capable of answering questions about visual data, which will be rigorously evaluated on a hidden dataset using an undisclosed metric.

Introduction

The challenging multimodal task of **Visual Question Answering (VQA)** requires models to comprehend both visual content (images) and natural language questions before providing precise responses. This project addresses the complexities of VQA by leveraging the diverse and detailed **Amazon Berkeley Objects (ABO)** dataset. We aim to create a practical multiple-choice VQA dataset, evaluate existing models, and innovate by applying **LoRA** for efficient and effective fine-tuning in resource-constrained environments.

Project Goals

Our primary objectives for this project are:

- **Dataset Generation:** To construct a comprehensive multiple-choice Visual Question Answering (VQA) dataset from the raw ABO dataset.
- **Baseline Establishment:** To evaluate the performance of already established VQA models as a baseline.
- **Efficient Fine-tuning:** To implement and apply **LoRA** for efficient fine-tuning of existing large pre-trained models.
- **Performance Optimization:** To achieve maximum growth in accuracy and performance on the VQA task within given constraints.
- **Robust Evaluation:** To evaluate the fine-tuned models using both standard and proposed metrics.

Amazon Berkeley Objects (ABO) Dataset

The dataset contains **147,702 product listings** (metadata for the corresponding images), incorporating **multilingual metadata** alongside **398,212 unique catalog images**. This version is the smaller **3GB** release (compared to the original **100GB**), which includes product **metadata in CSV format** and **images resized to 256x256 pixels**. This smaller version is crucial for quick experimentation and development without compromising data quality.

Dataset Access: Access ABO Dataset

Dataset Overview: Amazon Berkeley Objects (ABO)

The Amazon Berkeley Objects (ABO) dataset is a comprehensive repository featuring diverse product images under various viewpoints and lighting conditions, alongside detailed metadata. It serves as an ideal foundation for training and evaluating advanced VQA models. For this project, two primary `.tar` files were essential:

`abo-images-small.tar` (3 GB)

This archive contains downscaled (maximum 256 pixels) catalog images and their associated metadata. The downscaling is crucial for efficient storage, faster processing, and suitability as input for many deep learning models without significant loss of relevant visual information.

- `abo-images-small/images/metadata/images.csv`:
  - A gzip-compressed CSV file containing image metadata.
  - **Columns/Fields**:
    - `image_id`: Unique ID for each image.
    - `height`: Height of the image in pixels.
    - `width`: Width of the image in pixels.
    - `path`: The relative path to the image within the `images/small/` directory, using a hierarchical structure based on the first two characters of the image ID. Most images are in `.jpg` format; a few are in `.png`.
- `abo-images-small/images/small/`:
  - Directory containing actual image files, organized using a hexadecimal-based directory structure.

`abo-listings.tar` (83 MB)

This archive comprises product listings and their corresponding metadata, providing rich textual context for the images.

- `abo-listings/listings/metadata/`:
  - Contains 16 JSON files (`listing_1.json` to `listing_f.json`) with multiple unwrapped JSON objects.
  - Each object represents a product listing with up to 28 descriptive fields.

- **Key Fields for VQA Generation**:
  - `bullet_point`: Major product features (e.g., "Lightweight, Water resistant")
  - `color`: Named color (e.g., "green", "blue")
  - `color_code`: HTML color code (e.g., "#0000FF")
  - `fabric_type`: Describes fabric (e.g., "Cotton, Polyester")
  - `finish_type`: Describes finish (e.g., matte or glossy)
  - `item_shape`: Product form (e.g., "square", "rectangle")
  - `material`: Material description (e.g., Plastic, Metal)
  - `pattern`: Product design (e.g., Striped, Floral)
  - `product_description`: Textual HTML-embedded description
  - `product_type`: Product category (e.g., CELLULAR_PHONE_CASE)
  - `style`: Style label (e.g., Modern, Vintage)
  - `main_image_id`: Identifier for the main image
  - `other_image_id`: List of additional image identifiers

Workflow: From Raw Data to Curated Dataset

The data curation process was carefully designed to transform the raw ABO data into a structured dataset suitable for VQA model training.

1. **Initial Data Processing & Filtering**
   - Parsed the 16 `.json` files within `abo-listings/listings/metadata/`, which contained multiple JSON objects not encapsulated in a list. These were first wrapped into a standard JSON list for programmatic access.
   - Removed entries with non-English language tags (i.e., `language_tag` $\neq$ `en_US`) to avoid misinterpretation by the Gemini model during Q-A pair generation. An additional language check was also incorporated into the prompt.
   - Filtered out irrelevant fields, retaining only the 12 critical fields (e.g., `bullet_point`, `color`, `material`, `product_type`, `main_image_id`, `other_image_id`) to reduce noise.
2. **Image and Metadata Integration**
   - Merged `main_image_id` and `other_image_id` into a new `all_image_id` field, creating a comprehensive list of image associations per listing.
   - Queried `abo-images-small/images/metadata/images.csv` for each `image_id` to retrieve the corresponding `path`, which was joined with the root directory `abo-images-small/images/sma` to form complete image paths.
3. **Product Type Categorization & Distribution**
   - Organized the filtered JSON objects into directories based on the `product_type` field, resulting in 576 unique product categories.

- Distributed these 576 category folders equally among three team members, assigning 192 categories each to enable parallel processing and efficient dataset creation.

4. **VQA Pair Generation with Gemini**

   - For each JSON object, selected an `image_id` from the `all_image_id` field and retrieved the corresponding image using the constructed path.
   - Passed the image and relevant JSON metadata (excluding `all_image_id`) to the Gemini model through a carefully engineered prompt to generate high-quality question-answer pairs.

Product Type Distribution Analysis

(All the 12 parts are provided here: `/DataCuration/OriginalDataSummary/Plots`)

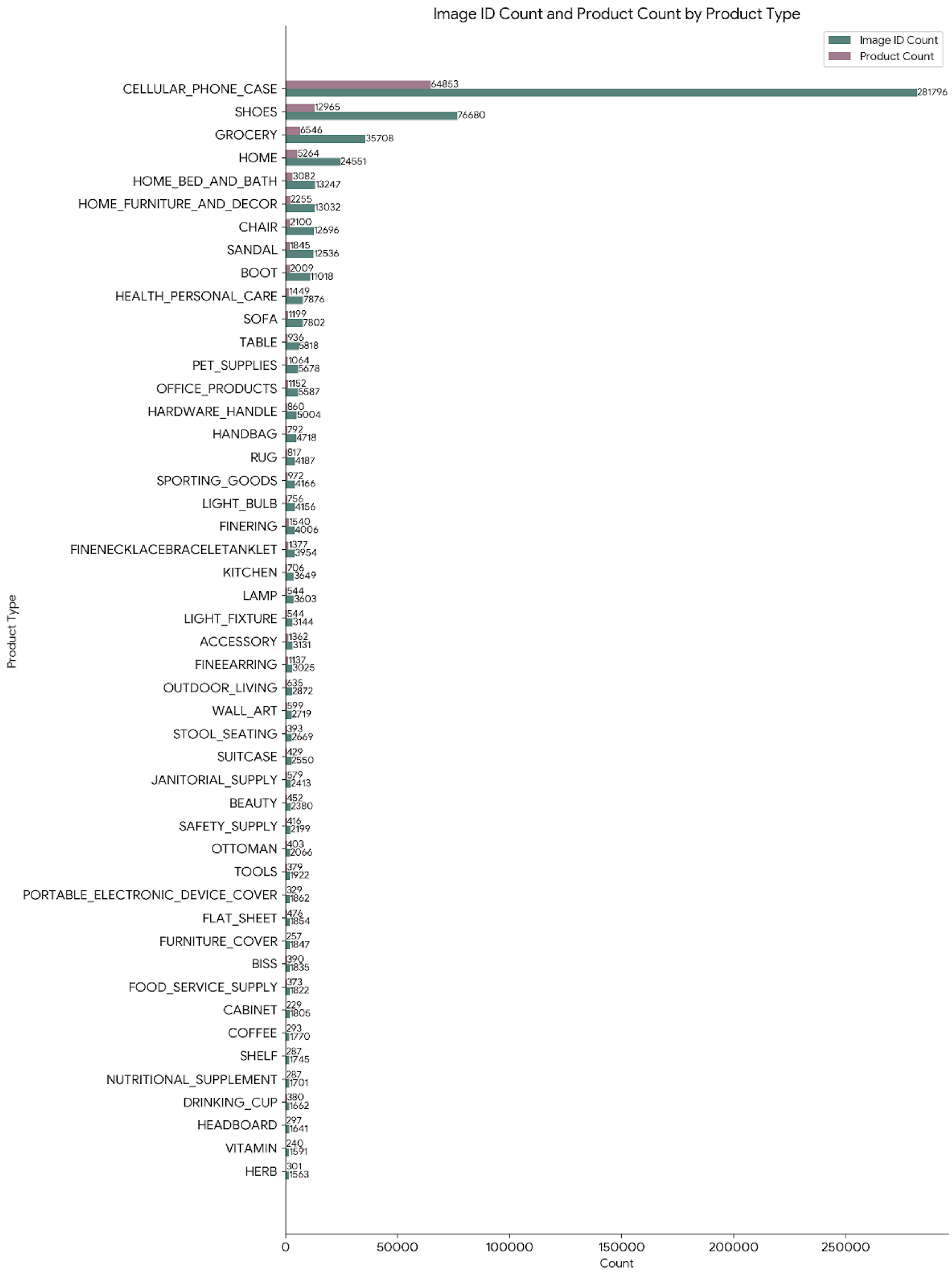


Figure 1: Group 1 Distribution

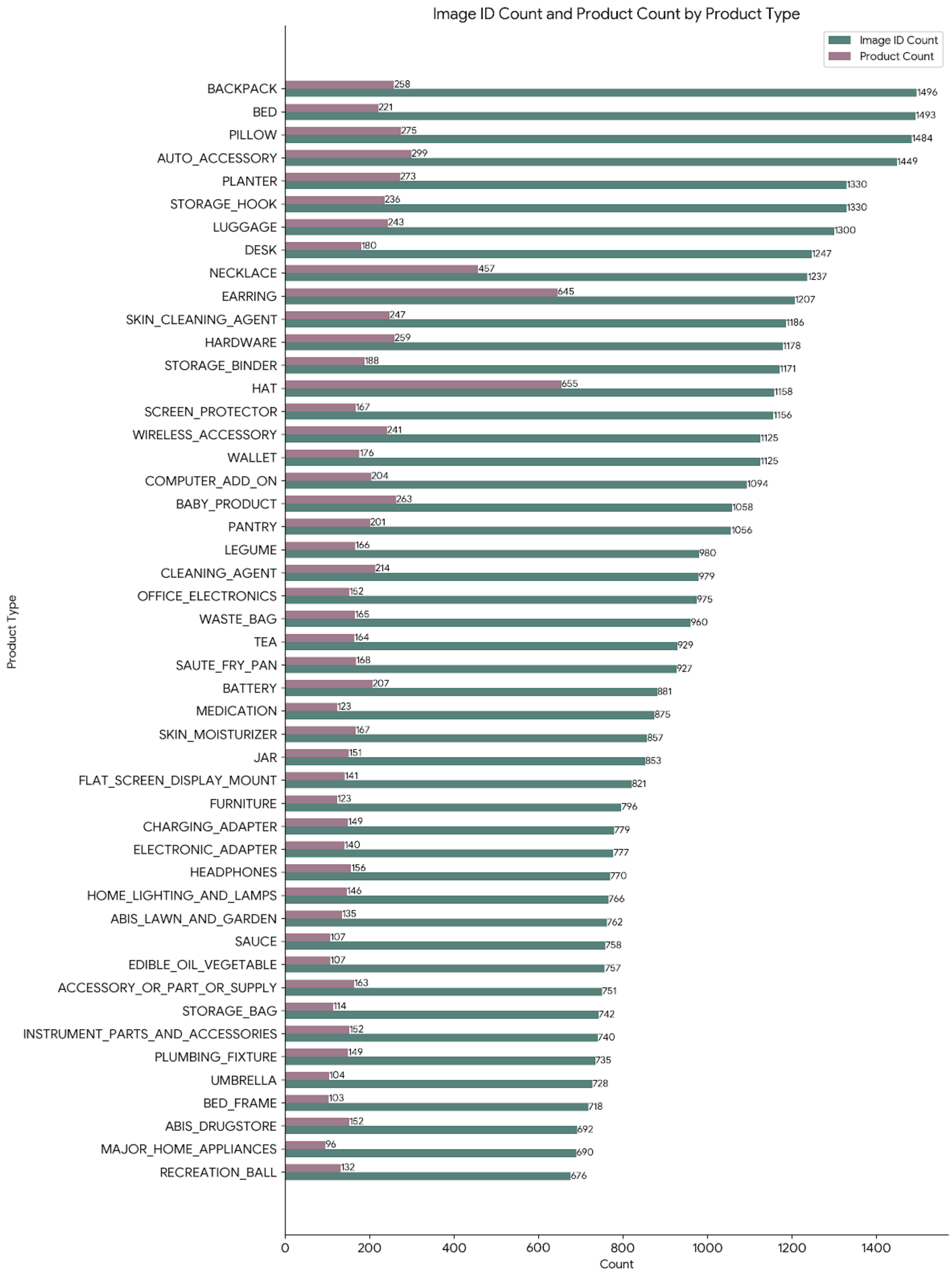


Figure 2: Group 2 Distribution

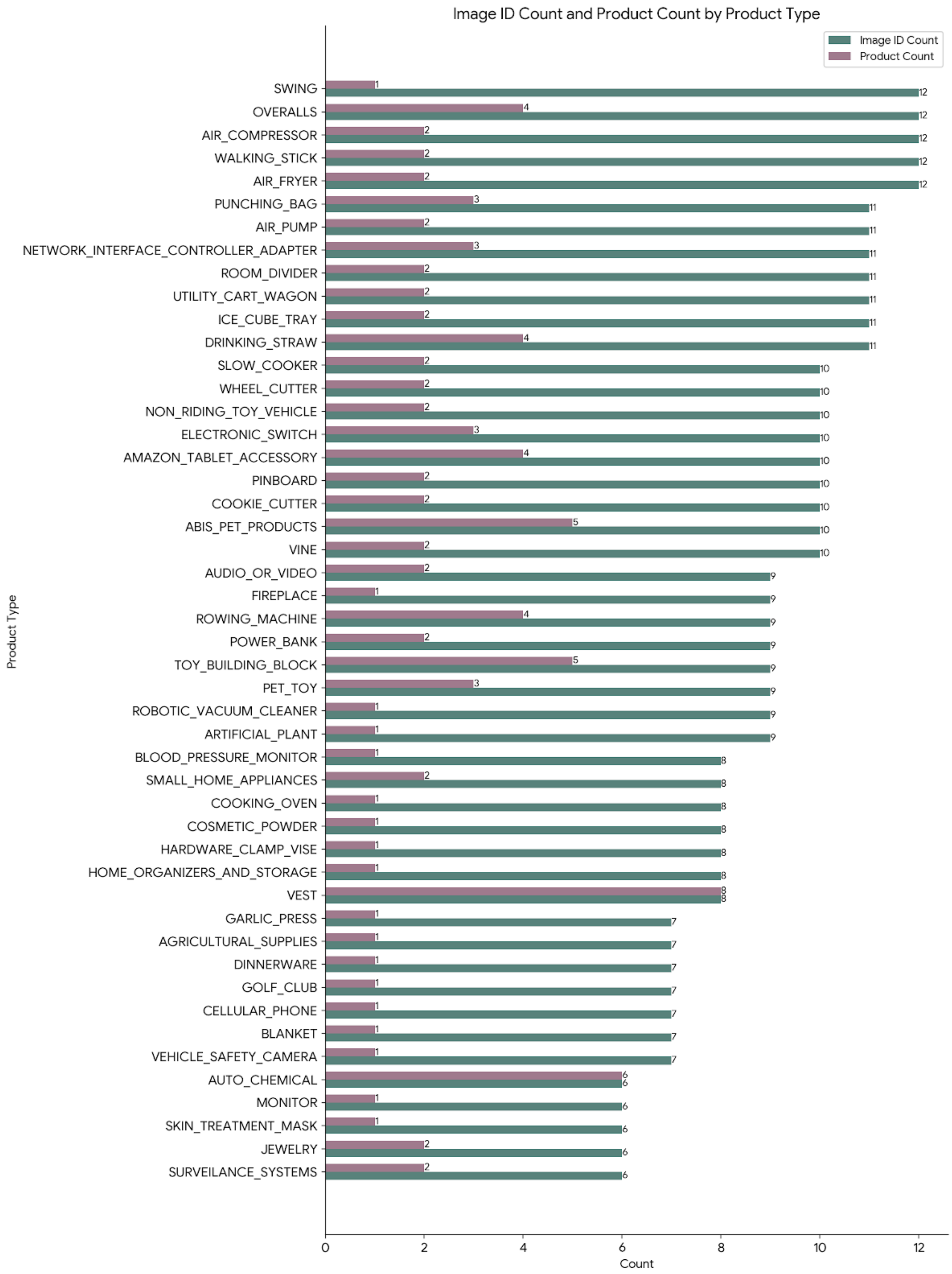


Figure 3: Group 11 Distribution

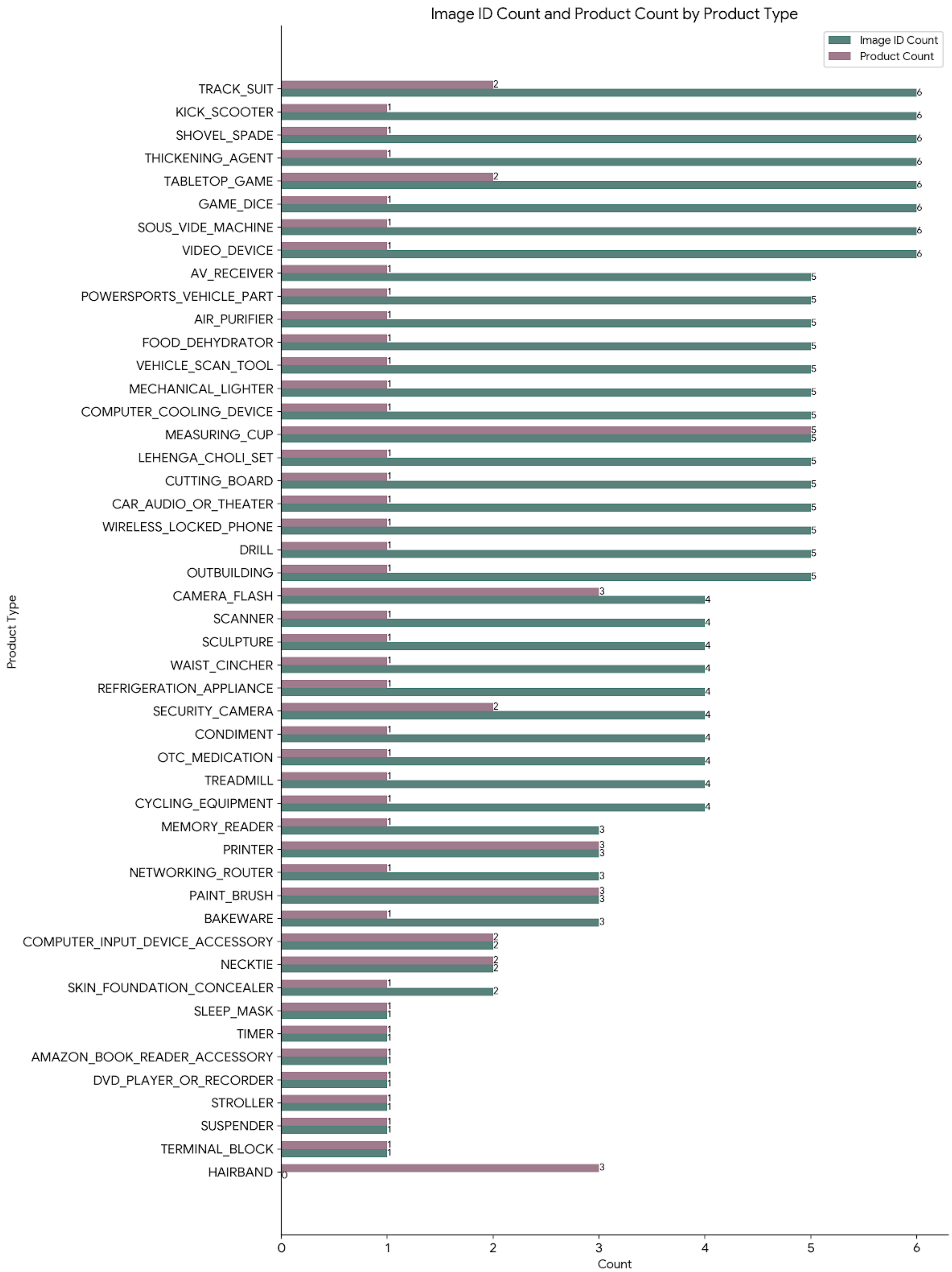


Figure 4: Group 12 Distribution

Here we have included the top 2 parts/groups and bottom 2 parts/groups. The remaining parts/group graphs can be accessed at the above-mentioned path.

Upon analyzing the distribution of the 147,702 product listings with multilingual metadata and 398,212 unique catalog images across the 576 product types, a severe class imbalance was observed.

Key Observations

- **Dominance of a Single Category:** The `CELLULAR_PHONE_CASE` category dominates the dataset overwhelmingly with about 64,853 listings and 281,796 images, representing almost 43% of the entire listing dataset and 70.7% of total images. This indicates a strong skew where one product type covers a large proportion of the data.
- **Extreme Long Tail:** Conversely, many product categories have fewer than 5 listings each, accounting for a combined total of only 0.0033% of the entire listing dataset or close to 0% of total images.
- **Implications:** These radical imbalances in category representation can introduce substantial bias during model training. A model trained on this unbalanced dataset would perform well on highly represented classes (such as `CELLULAR_PHONE_CASE`) but underperform on categories with few samples, undermining its ability to generalize and maintain fair performance across diverse product types.

In order to create a more representative and balanced dataset, it was important to implement a strong sampling strategy to normalize the product type distribution, ensuring diversity and reducing model bias.

Sampling Strategy: Proportional Tiered Sampling Algorithm

The extreme class imbalance — where a tiny group of classes (e.g., `CELLULAR_PHONE_CASE` with 281,796 distinct images or 64,853 product listings) accounts for a huge majority, while most have between 1 and 2 samples — was a major challenge. To address this, we implemented a **Proportional Tiered Sampling Algorithm**. This method samples listings from every product type proportionally to their frequency, resulting in a balanced and representative training set without losing valuable data from any category.

Proportional Tiered Sampling Algorithm

Step 1: Define Category Tiers Based on Number of Images

Tier	Description	Number of Images per Category
Tier 1	Very Large	100,000 images or more
Tier 2	Large	10,000 to less than 100,000 images
Tier 3	Medium	1,000 to less than 10,000 images
Tier 4	Small	Less than 1,000 images

Table 1: Category tiers based on number of images

Tier	Sampling Ratio
Tier 1	Sample 5% (0.05) of images
Tier 2	Sample 20% (0.20) of images
Tier 3	Sample 50% (0.50) of images
Tier 4	Sample 100% or at least 100 images (oversample if needed)

Table 2: Sampling ratios assigned per tier

Step 2: Assign Sampling Ratios Per Tier

Step 3: Apply Maximum Sample Cap

To control the total dataset size, apply a **maximum sample cap** per category. For example:

$$M = 15,000$$

where M is the maximum samples allowed per category.

Step 4: Calculate Final Number of Samples per Category

For each category:

1. Determine its tier based on the total number of images.
2. Calculate base samples using the tier's sampling ratio.
3. Adjust sample size using a logarithmic scaling factor (optional) to balance sampling:

   $$Adjustedsamples = (\log_{10}(CategorySize) \times \alpha + SamplingRatio) \times CategorySize$$

   where α is a scaling factor (e.g., between 0.1 and 1) to fine-tune sampling.
4. Ensure a minimum number of samples (e.g., at least 1 sample per category).
5. Limit samples to the maximum cap M.
6. For Tier 4 categories (small categories): If category size is less than 100 images, oversample to reach 100 samples.

Parameters to Customize

- Minimum guaranteed samples per category: e.g., 1
- Minimum fixed number for small categories: e.g., 100
- Maximum sample cap per category: e.g., 15,000
- Logarithmic scaling factor α: e.g., 0.5

Summary Table

This approach generates a more balanced and diverse dataset, which is important in training a strong VQA model that will perform well on all product categories, rather than merely the most frequent ones.

Tier	Category Size Range	Sampling Ratio	Minimum Samples	Maximum Samples
Tier 1	$\geq 100,000$	5%	1	15,000
Tier 2	10,000 to $< 100,000$	20%	1	15,000
Tier 3	1,000 to $< 10,000$	50%	1	15,000
Tier 4	$< 1,000$	100% or 100*	100*	15,000

Table 3: Summary of sampling strategy by tier

1 Prompt Engineering for VQA Generation

The prompt, stored in `/DataCuration/MainCode/Prompt.txt`, was carefully designed to create high-quality question-answer pairs. Its primary goal was to ensure full coverage of both visual information and product metadata, enabling the generation of varied, non-duplicate, and descriptive questions.

Key Considerations in Prompt Design

- **Complete Understanding**: The prompt guides the Gemini model to integrate information from the image (visual features such as shape, color, and pattern) with the provided metadata (e.g., `bullet_point`, `material`, `product_description`).
- **Diversity of Question Types**: Questions explore a wide range of aspects, including:
  - *Object Recognition*: *"What is the central object in the image?"*
  - *Spatial Relationships*: *"Where is the handle on the bag?"*
- **Material Properties**: Questions like *"What material is the product constructed of based on the description?"*
- **Dimensional and Quantitative Details**: Probing specific measurements or counts, e.g., *"What are the approximate sizes of the product?"* or *"How many compartments does the bag have?"*
- **Functional and Stylistic Attributes**: For example, *"What is the style of this product?"* or *"What are some of the main features listed in the bullet points?"*
- **Contextual Accuracy**: The prompt instructs the model to generate evidence-based questions and answers directly supported by the image and metadata, minimizing hallucinations.
- **Non-Redundancy**: It encourages generation of unique and distinct questions for each image to avoid repetition.
- **Detail and Specificity**: The questions aim for a deeper understanding beyond basic visual identification to infer properties or relationships.

This rigorous prompt design ensures the resulting dataset is balanced, context-rich, and effective at challenging VQA models, thereby improving their performance on real-world multi-modal tasks with greater scope and precision.

2 Model Choice: Llama or Gemini

One of the most important decision-making steps was whether to use the Llama or Gemini model for creating our Q&A pairs. To make a well-informed decision, we conducted an experimental assessment using 30,000 images with both models and manually evaluated the quality of the generated Q&A pairs. We also employed the BLIP model as a baseline to estimate accuracy metrics.

Llama Model

Advantages:

- **Local Installation**: One great benefit was that it could be installed and executed locally, without the need for external API keys and with the option to operate continuously, 24/7, without interruption.

Cons:

- **Quality**: In contrast to its operational adaptability, we found a precipitous decline in the quality of the Q&A pairs generated by this model when compared to Gemini. The answers and questions were less accurate in terms of context and less rich in content.

Gemini Model

Advantages:

- **Higher Quality**: The Gemini model uniformly outperformed others in generating high-quality, contextually rich Q&A pairs.
- **Elevated Capabilities**: Not only did it perform admirably in default question generation but also demonstrated extraordinary abilities at extracting exact measurements from images and creating applicable Q&A pairs from these accurate attributes. This was a major distinguishing factor.

Cons:

- **API Key Dependence & Rate Limitations**: One of its main limitations was its dependence on API keys, which imposed usage caps. In particular, with one API key, the model could only produce about 15 Q&A pairs for every 750 images processed. This limitation arises because the high-quality output of Gemini requires more descriptive and computationally intensive analysis per image, especially for sophisticated questions such as measurement extraction or contextual relationships.

Decision and Rationale

After thoroughly evaluating both models, we ultimately chose **Gemini** over **Llama** for our Q&A task. Despite the challenges posed by API key limitations and generation rates, Gemini's unparalleled ability to produce high-quality, contextually accurate, and detailed Q&A pairs—including the precise extraction of measurements—provided a significant value proposition that outweighed the operational conveniences of Llama. Its advanced capabilities were deemed essential for achieving the project's goal of a top-tier VQA dataset.

To address Gemini's API limitations and efficiently scale the generation process while maintaining quality, we developed a novel API key management and error handling strategy, which is detailed in the subsequent section.

3 Implementation Details: The Code (/DataCuration/MainCode/Final.py)

Our robust implementation was designed to manage API calls efficiently, handle errors gracefully, and ensure the continuous generation of Q&A pairs for a large dataset.

API Key Management and Error Handling Strategy

- **API Key Set**: We maintain an `api_key.txt` file containing a set of 8–10 API keys. An `.env` file tracks the currently active API key. The active key is always positioned at the beginning of the list in `api_key.txt` and accessed via the `.env` file.
- **Errors Handled**: During execution, we encountered typical API-related errors, including:
  - `quota exceeded`
  - `connection reset`
  - `service unavailable`
  - `deadline exceeded`
  - `unavailable`
  - `504 Gateway Timeout`
  - `503 Service Unavailable`
  - `429 Too Many Requests`

  These errors generally occur around processing 21 images consecutively, indicating temporary server sleep or rate limit triggers.
- **Retry Mechanism**: To mitigate these issues, our system employs a sophisticated retry logic:
  1. Upon encountering an error, the process waits for 30 seconds.
  2. It attempts up to 5 retries with 30-second waits between each retry on the current API call.
  3. If all 5 retries plus the initial attempt fail (6 attempts total), the current API key is considered exhausted.
  4. The exhausted API key is moved to the bottom of the `api_key.txt` list, and the process continues with the next key at the top of the list, enabling seamless cycling through the API key pool.

Output Storage

- The Q&A pairs are persistently cached in a separate directory, stored as individual `.json` files.
- Each `.json` file is named after its associated `image_id`.
- Each file contains two primary fields:

  - `image_path`: Relative path to the image file.
  - `qa_pairs`: A list of dictionaries, each representing a Q&A pair generated for that image.

This robust implementation plan enabled us to tame the complexities of API rate limits and intermittent service disruptions, ensuring efficient and uninterrupted production of a large-scale VQA dataset.

Results and Dataset Statistics (`/DataCuration/FinalGeneratedDataSummary/plots`)

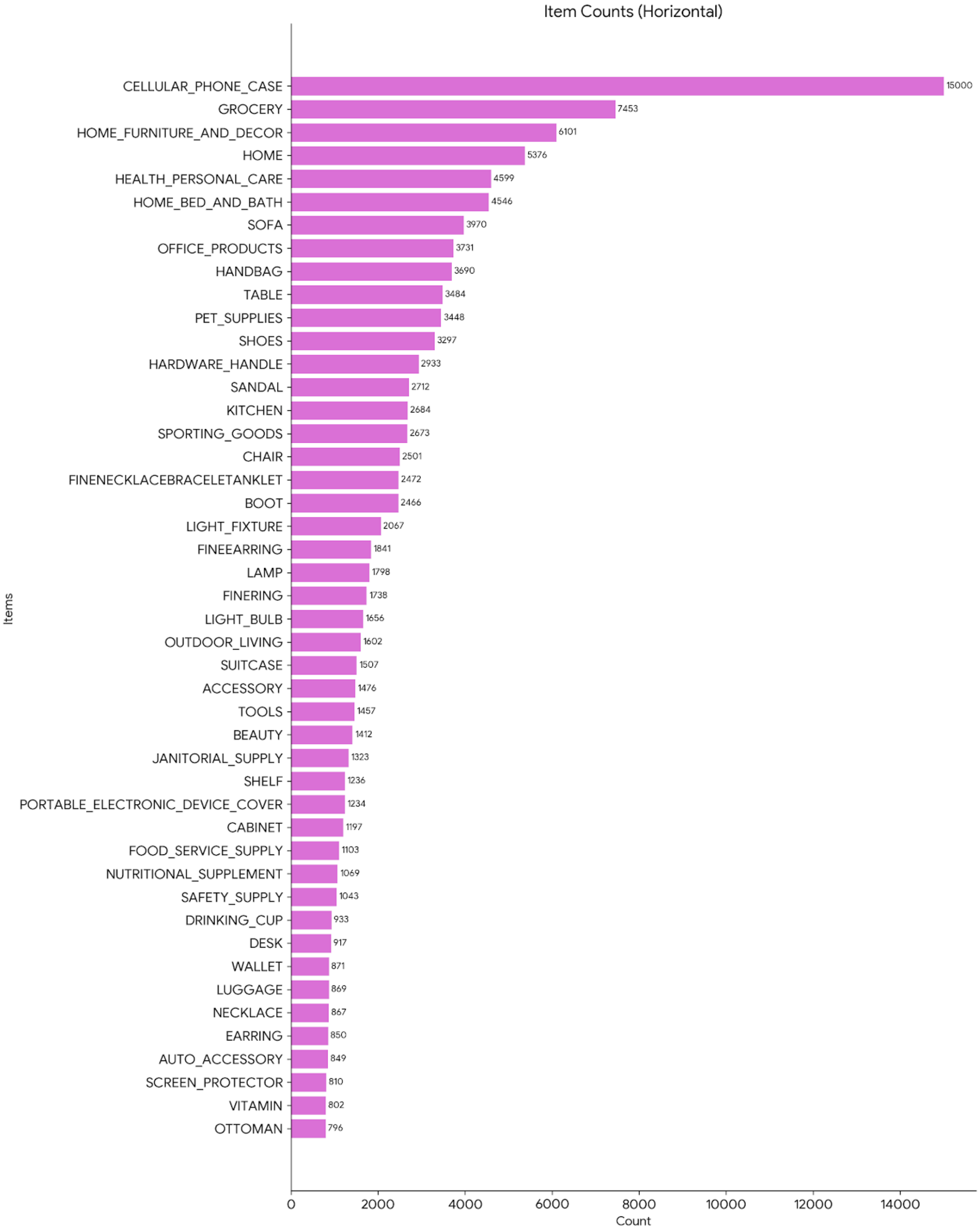


Figure 5: Group 1 Distribution

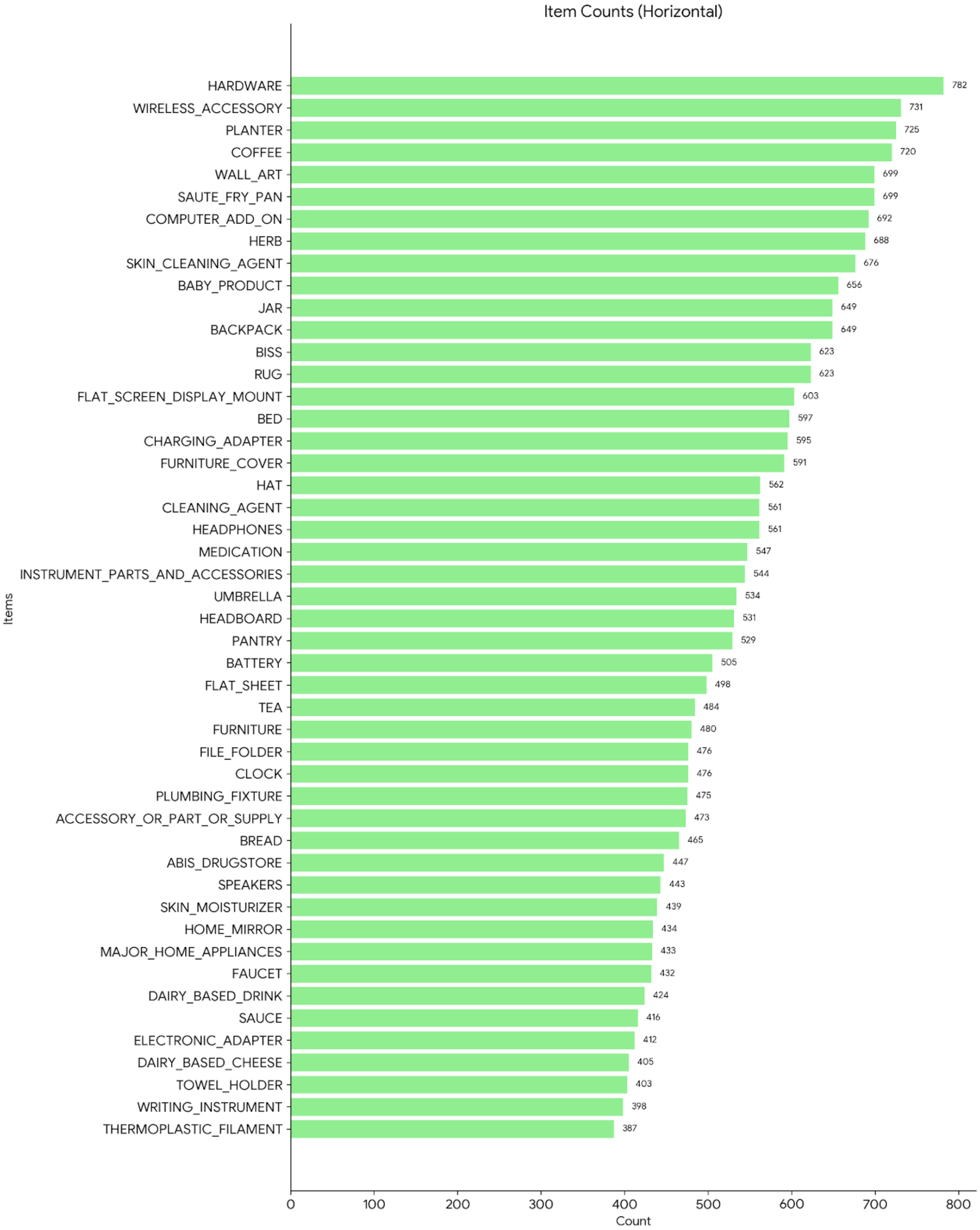


Figure 6: Group 2 Distribution

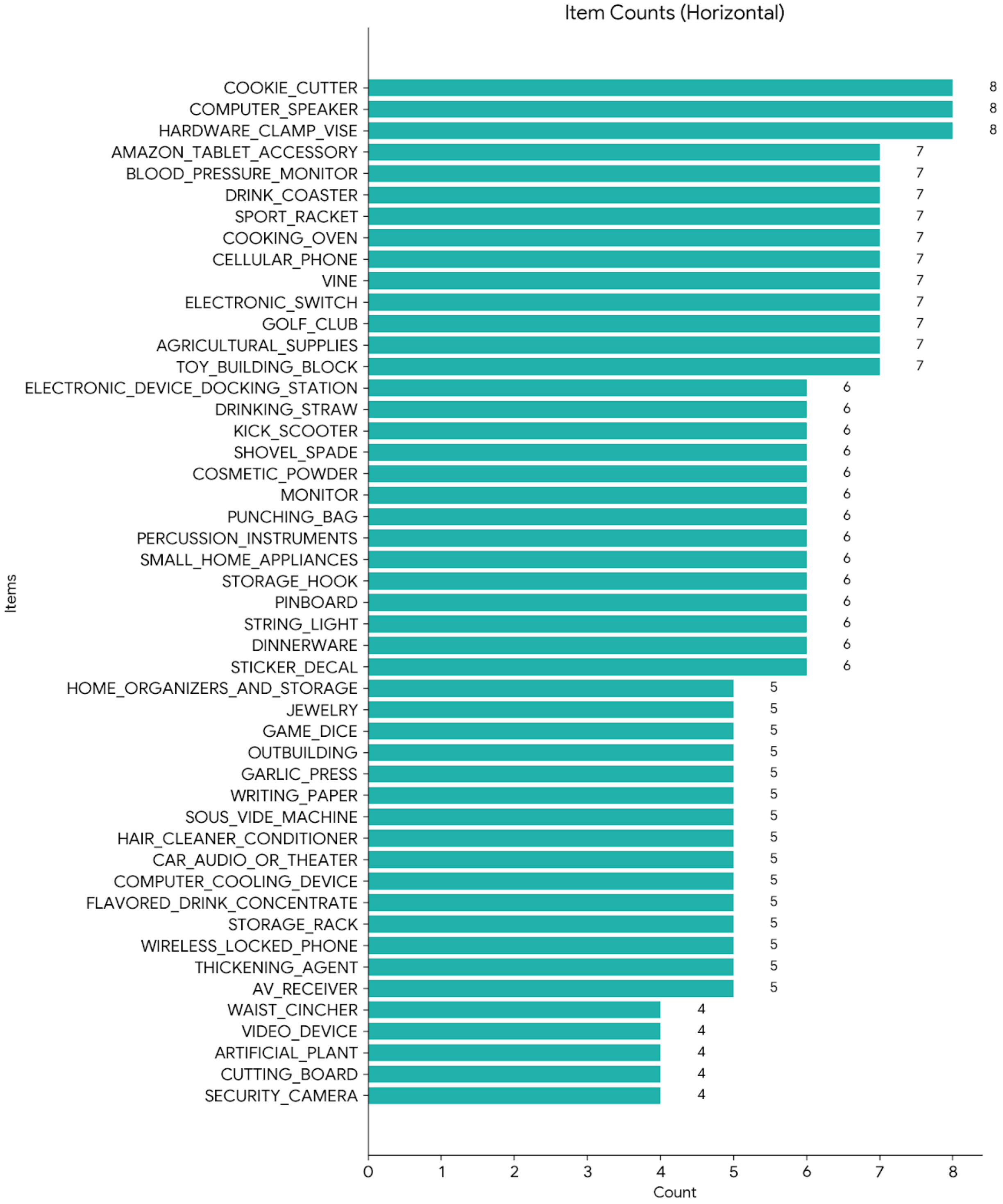


Figure 7: Group 11 Distribution

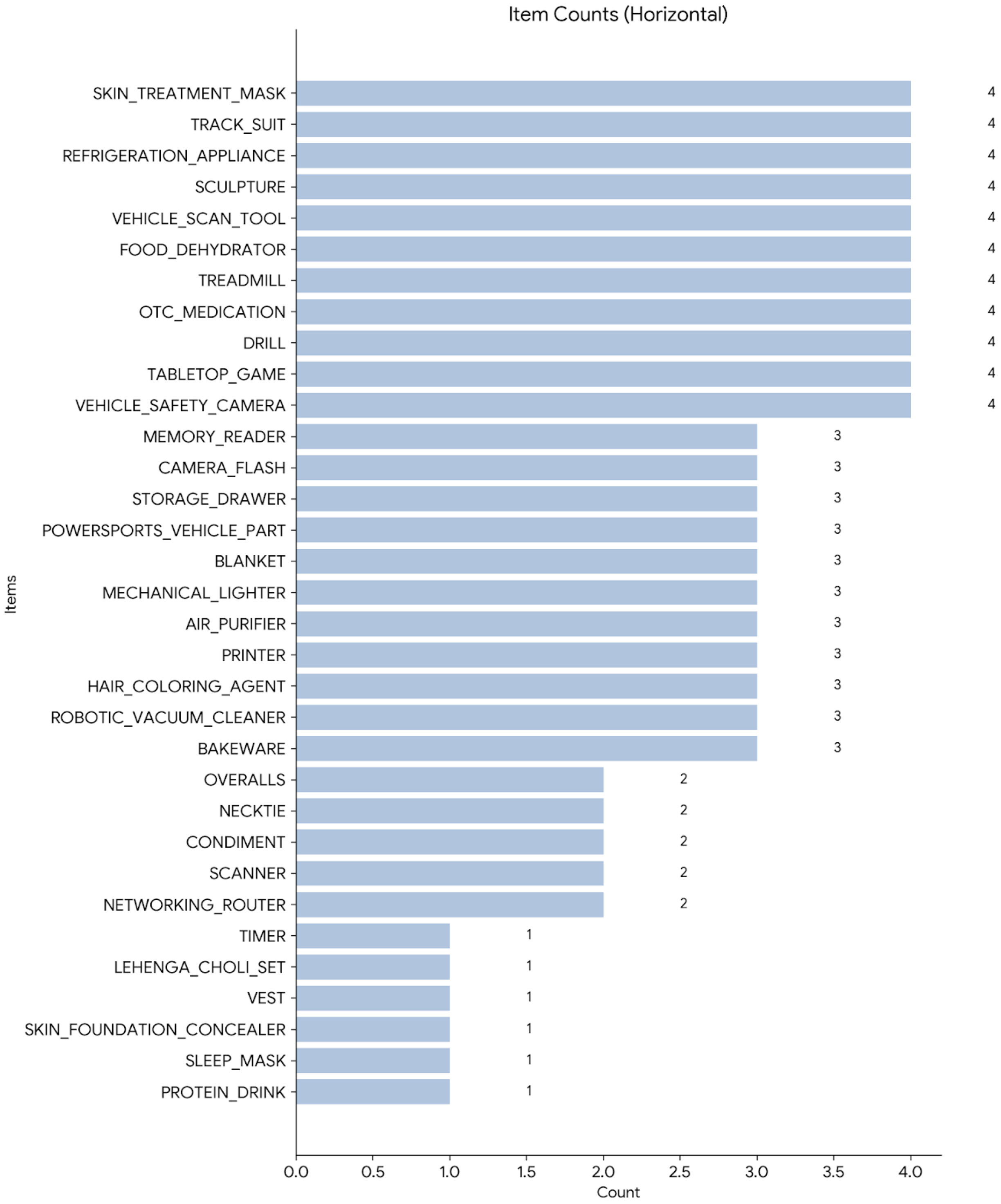


Figure 8: Group 12 Distribution

The remaining parts/groups graphs can be accessed at the above-mentioned path.

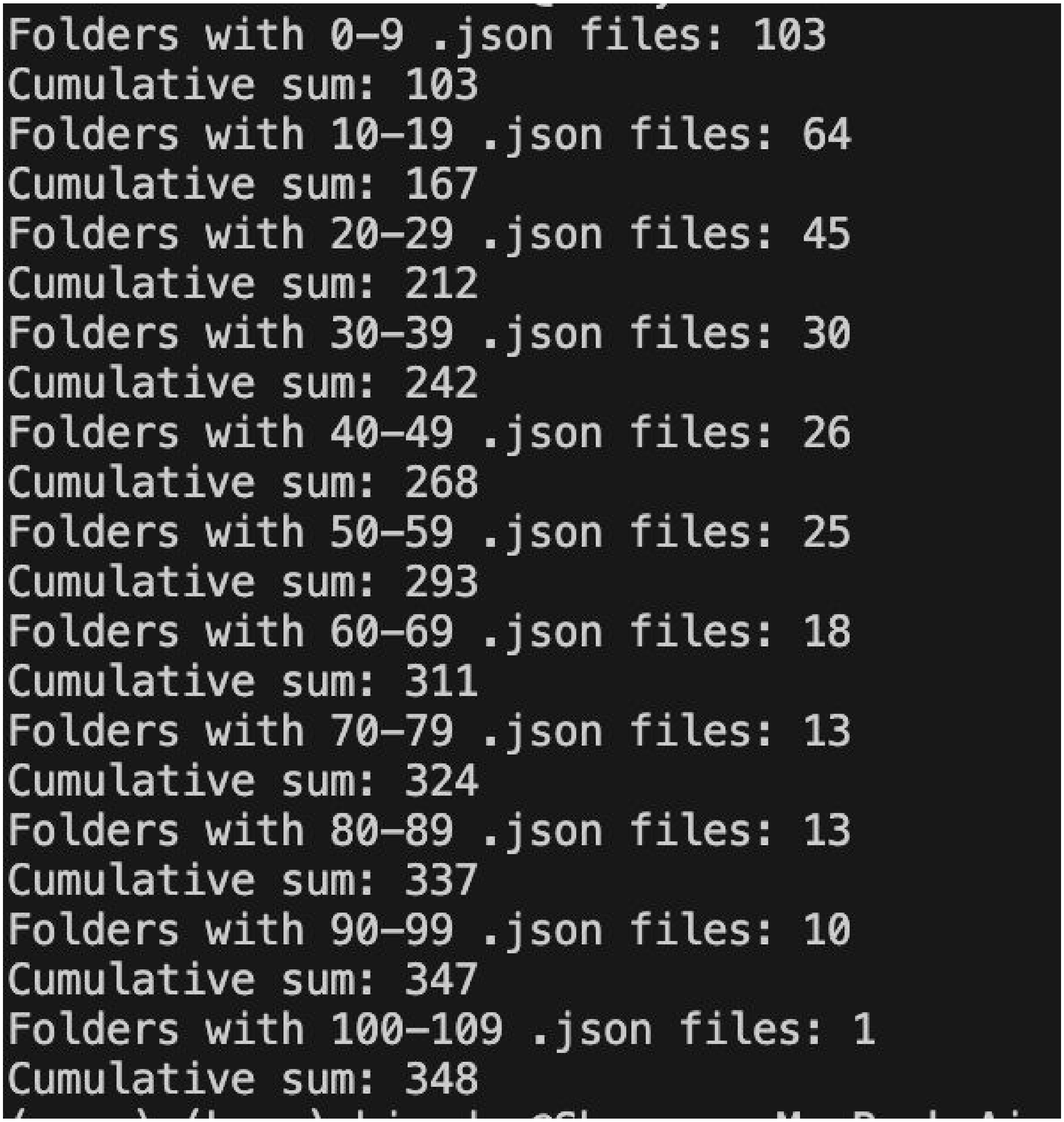
We successfully generated question-and-answer (Q&A) pairs for **169,659 images**, distributed across **559 categories**.

- **Category Reduction**: We observed a slight reduction in the total number of categories from the initial 576. This is attributed to multiple `image_id`s potentially sharing the same `product_type` but not being sampled, or some categories being excluded by the sampling algorithm if their count was extremely low and did not meet the minimum threshold after processing.
- **Fairness in Distribution**: Crucially, the implemented sampling algorithm ensured that the number of images within each category was sampled properly according to our proportional tiered strategy, maintaining a fair and balanced representation across the diverse product types. The detailed summary of product types and their corresponding counts can be found in `/Data Curation/FinalGeneratedDataSummary/category_counts.txt`.

Train-Test Split Strategy

The train-test split was an essential step in developing strong training and test sets. The major challenge was to preserve high quality and sufficient samples for effective evaluation of the test dataset, particularly given the initial class imbalance.

Choosing a Threshold



```
Folders with 0-9 .json files: 103
Cumulative sum: 103
Folders with 10-19 .json files: 64
Cumulative sum: 167
Folders with 20-29 .json files: 45
Cumulative sum: 212
Folders with 30-39 .json files: 30
Cumulative sum: 242
Folders with 40-49 .json files: 26
Cumulative sum: 268
Folders with 50-59 .json files: 25
Cumulative sum: 293
Folders with 60-69 .json files: 18
Cumulative sum: 311
Folders with 70-79 .json files: 13
Cumulative sum: 324
Folders with 80-89 .json files: 13
Cumulative sum: 337
Folders with 90-99 .json files: 10
Cumulative sum: 347
Folders with 100-109 .json files: 1
Cumulative sum: 348
```

Figure 9: Threshold Distribution

Upon analyzing the image distribution, we observed a clear pattern indicating that categories with fewer than a certain number of images would not provide statistically significant evaluation. Consequently, we established a threshold of **20 images**.

Data Splitting Methodology

1. **Training Set**:
    - Included all product types, regardless of their image count. This ensures that the training model is exposed to the full diversity of product categories, even the rare ones.
2. **Test Set**:
    - Only product types with **more than 20 images** were considered eligible for inclusion in the test set. This guarantees that each category represented in the test set has a minimum level of statistical significance for evaluation.
    - From these eligible product types, **only 20% of their images** were randomly selected to form the test set. This ensures a manageable test set size while maintaining representativeness.

Dataset Statistics

Following this strategy, we successfully extracted:

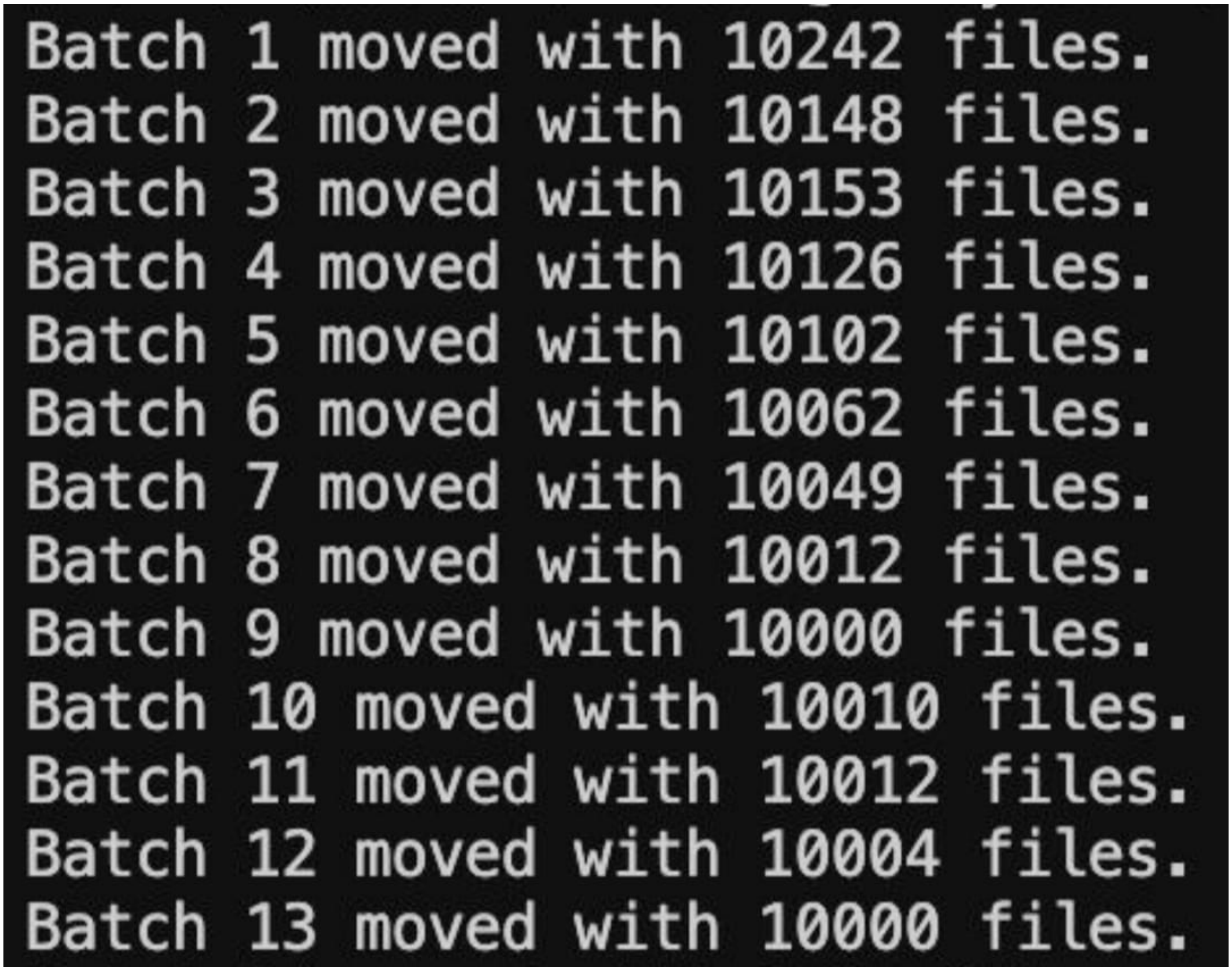
- **34.1k JSON files** containing question-and-answer (Q&A) pairs for the **test set**.
- The remaining **135k JSON files** formed the **train set**.

Screenshots illustrating this split can be found in `/Data Curation/train-test-split`.

Batching Strategy for Training Data

Given the substantial size of the training dataset (135k JSON files), storing them all in a single directory was not feasible for efficient model training and data loading. Therefore, we implemented a structured batching process:

- **Even Distribution**: The process evenly distributes '.json' files from multiple category folders (within the `Batches/` directory, where the data generation output was stored) into smaller, manageable batches.
- **Batch Size**: Each batch was capped at approximately **10,000 files**.
- **Round-Robin Approach**: To ensure each batch contained a balanced mix of files from all available categories and prevented class imbalance within individual batches, a round-robin approach was used. The system first collected all '.json' files from each category folder, shuffled them to introduce randomness, then iteratively built batches by taking one file at a time from each category in rotation.
- **Directory Structure**: Once a batch reached roughly 10,000 files, it was moved into a newly created folder under `master_train/`. This process continued until all files were distributed, maintaining class diversity and preventing skewness in any single batch.



```
Batch 1 moved with 10242 files.
Batch 2 moved with 10148 files.
Batch 3 moved with 10153 files.
Batch 4 moved with 10126 files.
Batch 5 moved with 10102 files.
Batch 6 moved with 10062 files.
Batch 7 moved with 10049 files.
Batch 8 moved with 10012 files.
Batch 9 moved with 10000 files.
Batch 10 moved with 10010 files.
Batch 11 moved with 10012 files.
Batch 12 moved with 10004 files.
Batch 13 moved with 10000 files.
```

Figure 10: Batch Wise Distribution

This comprehensive train-test splitting and batching strategy ensures a robust evaluation framework and an efficiently loadable training corpus for developing VQA models.

Final Dataset Overview

We've carefully curated two high-quality datasets (`master-train` and `master-test`) to ensure robust model training and evaluation. Each dataset is hosted on Kaggle and publicly accessible.

Training Set – `Master-Train`

- Access Master-Train Dataset
- Access Master-Train (CSV) Dataset

Test Set – `Master-Test`

- Access Master-Test Dataset
- Access Master-Test (CSV) Dataset

4 Model Fine-tuning with LoRA

Model Fine-tuning with LoRA

In order to make a general Visual Question Answering model perform well on our target domain, an essential task was fine-tuning it on the relevant data. This implies modifying the parameters of the model so that it can grasp the particular visual and linguistic patterns of the new task better. To make this fine-tuning efficient and resource-conscious, we used Low-Rank Adaptation (LoRA). LoRA is a cutting-edge method that updates only a subset of weights of the model and gives a very efficient method to fine-tune big models like BLIP For Question Answering at a fraction of the cost of full fine-tuning. This section presents the methodology and implementation details of using LoRA on our pre-trained model.

Base Model and Processor

The foundation of the fine-tuning process was the `Salesforce/blip-vqa-base` model and its corresponding processor, loaded from Hugging Face Transformers.

Model Choices: Rationale for selected models and any alternatives considered

The main reasoning behind this decision is that **BLIP** is a state-of-the-art vision-language model pre-trained on large datasets for joint image-text understanding and generation, including the Visual Question Answering (VQA) task. It offers a solid, purpose-built foundation for fine-tuning. Though larger models such as `BLIP-2` variants (e.g., `blip-2.7b`, `blip-3bblip-7b`) are potentially more performant due to their higher parameter counts, they demand significantly more computational resources in terms of GPU memory and training time—critical factors given the hardware constraints typical in a course project setting.

During our evaluation, we observed that some larger models yielded promising results: for instance, `BLIP-2 3B` achieved an exact match accuracy close to 30% on our curated test set, and `BLIP-2 11B` pushed this to around 50%. We also explored other models like Bakllava and Qwen, which gave accuracies in the 20–30% range. However, these models presented similar computational challenges.

On the other hand, blip-vqa-base (350M parameters), though significantly lighter, achieved an exact match accuracy of around 14% on the same dataset. Despite the lower performance, its efficiency made it an ideal candidate: it required considerably less training time and resources, making it more feasible within the constraints of our project. Additionally, we anticipated that the heavier models—already performing well—might not yield proportionally higher gains post fine-tuning, especially given our dataset size and time limitations.

Alternatives such as CLIP, while strong in contrastive image-text alignment and zero-shot classification, are primarily encoder-based and not inherently designed for generative tasks like VQA. Using CLIP for direct question-answer generation would have necessitated coupling it with an external language model, thereby increasing system complexity and fine-tuning requirements—something beyond our project's intended scope.

Hence, adopting blip-vqa-base with the parameter-efficient LoRA fine-tuning method offered the most balanced trade-off between performance, computational feasibility, and task alignment.

Data Preparation and Loading

Imperative training of the Visual Question Answering (VQA) model demands organizing and pre-processing the image-question-answer data into a suitable format compatible with the BLIP model architecture. To this end, a custom PyTorch Dataset class, titled VQADataset, was created.

The data is structured with image files kept apart from their respective question-answer annotations. Annotations are offered as JSON files. A single JSON file holds metadata of an image, such as the relative path of the image (image_path), and a list of question-answer pairs generated by ai (qa_pairs) pertaining to that image.

VQADataset Class

The VQADataset class then acts as the bridge between the raw dataset files and the PyTorch training/evaluation pipeline. It is mostly responsible for loading, pre-processing, and formatting the image-question-answer triplets in a form directly consumable by the BLIP VQA model. This tailored implementation gives flexibility in addressing our particular data structure and pre-processing needs.

The class is intended to carry out the following main functions:

- **Scanning Specified JSON Directories:** The dataset constructor (__init__) accepts a list of directories (json_dirs). It then recursively goes through these directories with os.walk to locate all files that end with the .json extension. This makes it possible to construct the dataset from a collection of distributed annotation files that might be grouped into various batches or subsets. Each JSON file found should have the VQA annotations for one image.
- **JSON Data Processing:** For each found JSON file, the class loads its contents with json.load. It retrieves the image_path, which usually includes a relative path to the associated image file, and the qa_pairs list, which contains several dictionaries, each of a single question-answer pair (question and answer keys) for the image.

- **Full Image Path Construction:** Dataset annotations tend to utilize relative paths. The `VQADataset` class accepts a `base_img_path` argument for the constructor. It forms the absolute path to the image file by joining the `base_img_path` with the relative `image_path` for each image entry in the JSON. A dedicated cleaning step is provided to address possible inconsistencies in the relative path string (e.g., stripping off a constant prefix such as `abo-images-small/` if applicable), so the built-up path suitably points to the image location within the image storage of the dataset.

- **Storing Image-Question-Answer Samples:** The central data structure in the dataset object is an item list called `self.samples`. Each of these items in this list is a dictionary of one VQA sample. The dictionary has the fully formed `image_path`, the `question` string, and the `answer` string for one particular question-answer pair that is relevant to that image. By converting the JSON structure into individual question-answer-image samples, we make the dataset very flexible in that each item directly maps to one prediction task for the model.

- **Preprocessing in `__getitem__`:** The `__getitem__` function is called by the PyTorch `DataLoader` to get and preprocess one sample at an index.

  - **Loading and Formatting the Image:** The image file identified by the sample's `image_path` is loaded into memory from disc using the Pillow library. It is then immediately transformed into the `RGB` format (`.convert("RGB")`). This will guarantee that all images are in a standard three-channel structure, which is typical input to most vision models, irrespective of their native format (e.g., grayscale).

  - **Multimodal Input Processing:** The BLIP processor (`processor` object, defined outside the class) is the main vehicle for converting the raw question and image into numerical tensors for the BLIP model. The processor is responsible for both image processing (resizing and normalization according to the requirements of the pre-trained model) and text tokenization (converting the string question into a series of numerical token IDs). The `return_tensors="pt"` parameter guarantees the output will be PyTorch tensors. Truncation (`truncation=True`) and padding (`padding="max_length"`) are enforced on the tokenized question to guarantee all input sequences have the same length of `max_length=128`. This is a requirement for efficient batch processing by the model.

  - **Answer Tokenization (Labels):** The VQA example's appropriate answer string is tokenized independently with the tokenizer of the BLIP processor. Like the question, the sequence is padded and truncated to maintain a fixed length of `max_length=10` in the answer sequence. The tokenized answer sequence is the ground truth `labels` tensor for which the model will be trained to produce.

  - **Tensor Formatting:** The resulting tensors from the processor, which originally have a batch dimension of size 1 (as a result of `return_tensors="pt"` processing a single sample), are squeezed with `.squeeze(0)` to eliminate this dimension. This is the format to expect for single samples in a `Dataset` when utilized with a `DataLoader` to batch. The `inputs_embeds` tensor, if the processor generates it, is specifically removed since the BLIP model variant employed here accepts token IDs (`input_ids`) as input for the language component, rather than embeddings.

Through these steps, the `VQADataset` class successfully closes the gap between our raw data and the BLIP model and simplifies the fine-tuning procedure within the PyTorch and Hugging Face Transformers environments.

LoRA Configuration

We used the `peft` library of Hugging Face, which offers a convenient manner to use LoRA in Transformers library models.

The LoRA configuration used in fine-tuning was specifically selected to meet efficiency and performance needs:

- **`r` = 8:** This argument specifies the **rank** of the low-rank update matrices A and B. LoRA approximates the weight update matrix of a layer (ΔW) with the product of two significantly smaller matrices, A and B, such that $\Delta W \approx BA$. The dimensions of these matrices are $d \times r$ and $r \times k$ respectively, where $d \times k$ is the dimension of the original weight matrix W, and r is the rank. Increased rank permits a more expressive update but adds trainable parameters ($r \times (d+k)$). A ranking of 8 is a general starting point that tends to be well-balanced between parameter efficiency and capacity for capturing required adaptations.

- **`lora_alpha` = 32:** This is the **scaling factor** used for the LoRA updates. The scaled update to be added to the initial weight matrix is $(BA) \cdot (lora\_alpha/r)$. `lora_alpha` and `r` together set the size of the LoRA updates that are applied at fine-tuning time. Applying `lora_alpha = 32` with `r = 8` scales the updates by a factor of $32/8 = 4$. This scaling avoids the LoRA updates becoming too tiny, particularly when utilizing a lower rank.

- `target_modules` = [`'qkv', 'projection'`]: This indicates which modules (layers) of the pre-trained BLIP model the LoRA matrices are inserted into. `'qkv'` often denotes the linear layers that project the input embeddings into Query, Key, and Value vectors in the self-attention mechanism. `'projection'` probably denotes the output projection layer in the attention mechanism. Using LoRA in these attention-related layers is an effective and common approach since the attention mechanism plays a key role in the model learning about relationships in the input (text tokens and image features) and producing contextually appropriate outputs. Targeting fine-tuning on these layers enables the model to fine-tune its basic understanding and generation skills to the new data.

- **`lora_dropout` = 0.05:** This specifies the **dropout probability** used on the outputs of the LoRA update matrices (BA) during training. Dropout is a form of regularization in which, at each training step, a proportion of outputs from one layer are randomly zeroed out. Dropout on the LoRA updates prevents overfitting by weakening the model's dependence upon any one of the LoRA parameters, thereby pressurizing it into learning stronger representations. 0.05 is a small dropout value, meaning that this is a light level of regularization.

- **`bias` = `'none'`:** This option governs whether bias vectors in the target modules are fine-tuned as well. If `bias` is set to `'none'`, it indicates that only the weight matrices in the provided `target_modules` are updated through the LoRA updates (BA), with the bias vectors of the layers remaining frozen. This is the conventional approach in LoRA since fine-tuning just the weights tends to be adequate for performance gain, further promoting parameter efficiency.

This particular LoRA setup was selected to effectively modify the BLIP VQA model, concentrating trainable parameters on important attention mechanisms with balanced scaling and moderate regularization, thus allowing effective fine-tuning within computational constraints available.

Iterative Training Strategy

Due to the large size of the entire VQA dataset, fine-tuning the BLIP model in a one-pass over all data posed great computational and memory demands. In response to this, a multi-stage, **iterative training strategy** was adopted. This entailed splitting the dataset into smaller, manageable portions and training the model sequentially on these batches, incrementing the learning from prior stages.

The overall dataset was logically segmented into **13 distinct 'master' batches**. The fine-tuning process proceeded as follows:

1. **Initialization:** Training started either from the initial pre-trained `blip-vqa-base` model checkpoint (for the very first batch) or by resuming the weights and configuration of the fine-tuned model checkpoint saved at the end of the *previous* master batch's training. This enabled the model to learn and adapt further from its most updated state.

2. **Master Batch N training:** The model was fine-tuned in each iteration only on the data found within the present 'Master Batch N'. Training was achieved through a typical deep learning loop: for every batch of data from the `VQADataset`, a forward pass was done to get the predictions, the loss (cross-entropy, as usual for sequence generation tasks such as VQA) was calculated, backpropagation was used to calculate gradients, and the trainable parameters of the model were updated through the **AdamW optimizer** using a defined learning rate of `lr` = **10e-5** and trained for 1 epoch.

3. **Checkpointing and Model Versioning:** To ensure robustness against interruptions and to facilitate the iterative process, comprehensive checkpoints were saved regularly and versioned.

   - Periodically, after a set number of training steps (1000 steps), and definitively at the end of processing each master batch, the following were saved to a designated directory:

     - The model's fine-tuned weights and configuration files using `model.save_pretrained`.

     - The processor's configuration and tokenizer files using `processor.save_pretrained`.

     - A dedicated 'training state' file (`training_state.pt`) using `torch.save`. This critical file included the state of the optimizer (`optimizer.state_dict()`), the current training step counter (`step_counter`), and the history of recorded losses (`loss_history`). Saving the optimizer state and step counter is essential for seamlessly resuming training exactly where it left off, preventing loss of progress.

   - Each time a master batch's training was completed and saved, the output directory was named following a versioning convention: `/kaggle/working/model_latest_vN`. The 'vN' denotes the version number, indicating that the model within this directory has been trained up to and *including* the data from Master Batch 'N'.

   - These versioned model checkpoints were then curated and preserved in a **Kaggle dataset**. This single storage in a Kaggle dataset facilitated easy management of various iterations of the fine-tuned model, monitoring of progress across versions, and easy loading of a particular version as the beginning point for following training batches or for testing. The training state file (`training_state.pt`) was stored in the respective versioned model directory within the Kaggle working environment prior to being possibly added to the dataset.

- Access Blip-FineTunedModel-Versions - Blip-FineTunedModel-Versions

4. **Global Validation:** After the successful training on every single master batch, the optimized model's performance was tested on a **global test dataset**. The test set contained around **482,036 samples** (as per your evaluation cell output) and did not change during the course of the iterative process. Validation on an independent, large test set enabled to observe the model's ability to generalize and observe the cumulative effect of training on each subsequent batch of data on the entire problem space, as opposed to the performance on the batch currently being processed.

5. **Iteration Transition:** To transition from training on 'Master Batch N' to 'Master Batch N+1', the `load_path` for the next run was assigned the `save_path` of the recently completed run. This helped to ensure that training for the subsequent batch commenced with the model's weights and optimizer state after it completed training on the last batch. At the same time, the `json_root_dir` within dataset load configuration was modified to reference the directory holding 'Master Batch N+1' data.

This incremental training procedure on various batches of data permitted the model to progressively process and learn from the complete big dataset, with efficient management of memory and computational resources. Additionally, the end-of-batch and periodic checkpointing ensured robustness and allowed uninterrupted performance monitoring through the global validation step, with feedback into how the model progressively got better with additional exposure to more data in subsequent batches.

Performance Trends Across Iterations

In order to track the performance of the iterative fine-tuning procedure and see how the model's performance improved with each successive master batch of data exposed to it, we monitored major evaluation metrics on the constant global test dataset after each batch's training. This gives a transparent visualization of the learning trajectory.

The following chart shows the performance of two main evaluation measures: **Exact Match (EM)** and **BERTScore F1**, on the global test set after fine-tuning the model through to and including data from each individual master batch (from Batch 1 to Batch 14). The base `blip-vqa-base` model's performance **prior** to any fine-tuning is the first baseline.

As would be expected with fine-tuning success, the graph indicates a general **trend of increasing** both for the Exact Match (EM) and BERTScore F1 measures as more data is trained on in each successive batch.

- **Exact Match (EM)** is a direct measure of the model's capacity to output answers that exactly match the ground truth. A growing EM suggests the model is getting more accurate in factual recall and language generation for the target VQA task.

- **BERTScore F1** is a measure of semantic similarity between the generated answer and the ground truth answer, giving a better answer quality judgment. An increasing BERTScore F1 indicates that even if the generated answer is not an exact match, its meaning is increasingly approaching the meaning of the correct answer.

This increasing trend confirms our incremental training approach and proves that fine-tuning on

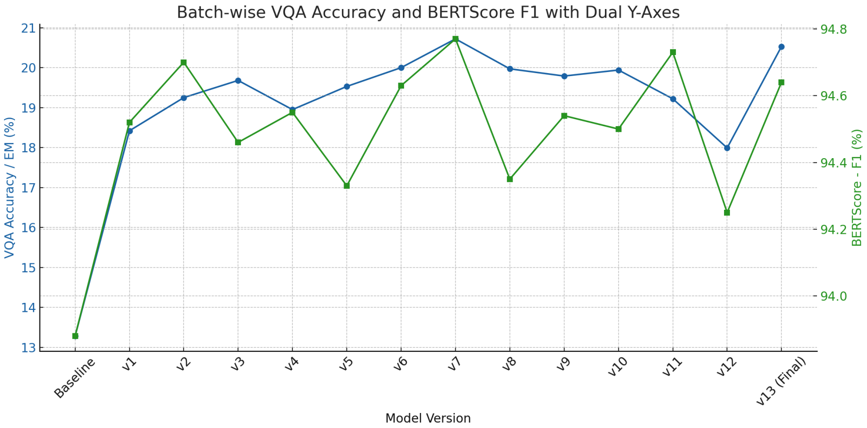


Figure 11: Performance Metrics Across Training Batches

every additional set of data added positively to the overall performance as well as generalization power of the model on the unseen test dataset. The starting performance (Batch 0 or Base Model) also acts as a good benchmark to indicate the gain achieved via the fine-tuning process.

Intra-Batch Training Dynamics: Loss Progression

Beyond tracking overall performance metrics across master batches, it was also crucial to monitor the training loss progression *within* each individual master batch. This offered insights into the learning stability, convergence speed, and allowed for early detection of potential issues like overfitting or learning plateaus during the fine-tuning on a specific data segment. The loss, typically cross-entropy for VQA tasks, was recorded at regular intervals (e.g., every few steps) throughout the training of each master batch.

Presenting the loss graphs for all 14 master batches would be too voluminous for this report. However, to illustrate the typical learning behavior observed, the loss progression graphs for Master Batch 6 are shown below as a representative example. Similar trends, with variations based on the specific data characteristics of each batch, were observed across other master batches.

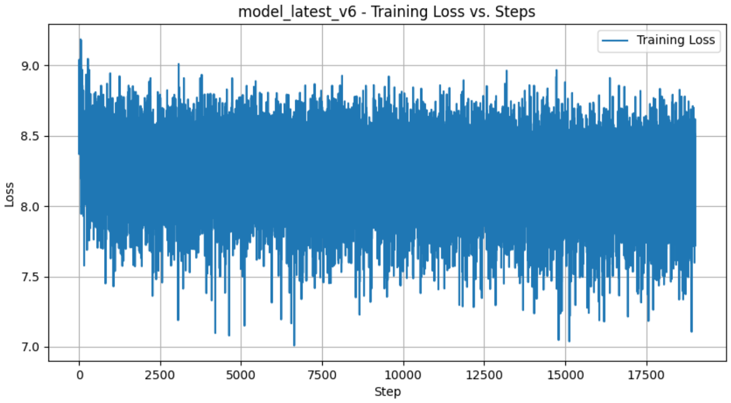


Figure 12: Raw Training Loss vs. Steps for v6 iteration.

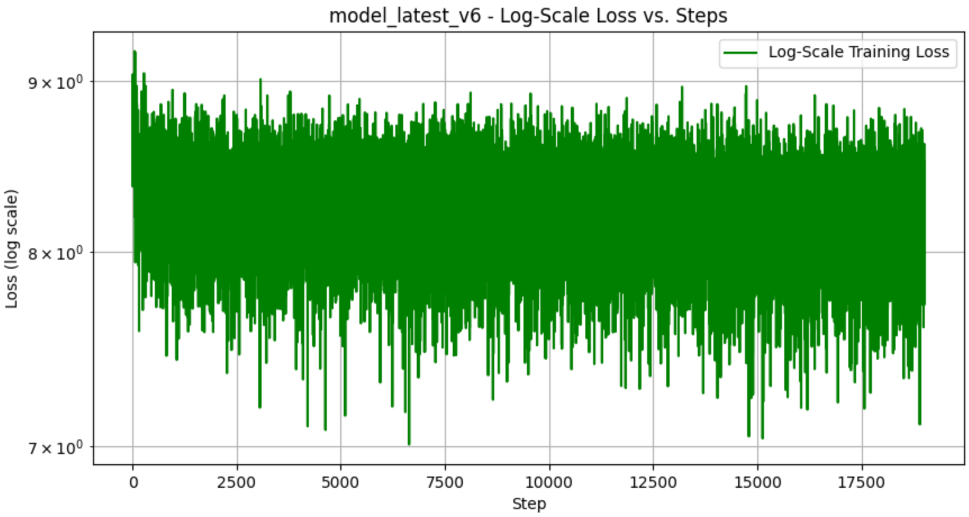


Figure 13: Log-Scale Training Loss vs. Steps for v6 iteration.

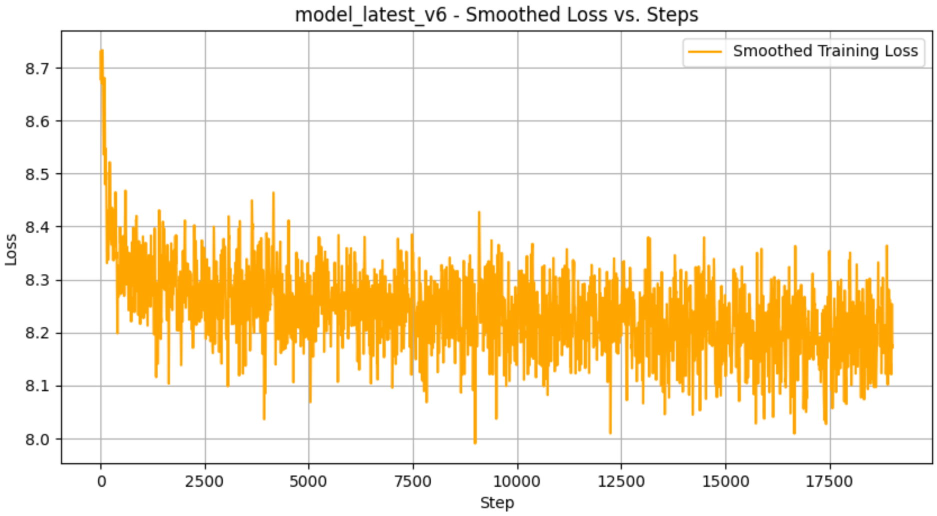


Figure 14: Smoothed Training Loss vs. Steps for v6 iteration.

The training loss graphs for model_latest_v6 (Figures 12, 14, and 13) demonstrate a characteristic learning pattern. Initially, all three representations of the loss show a steep decline in the early training steps, indicating rapid learning as the model parameters begin to adjust to the new data. Following this initial sharp drop, the rate of decrease slows, and the loss continues to trend downwards but with notable fluctuations, particularly visible in the raw (Figure 12) and log-scale (Figure 13) plots. The smoothed loss (Figure 14) confirms this overall downward trajectory while filtering out high-frequency noise, making the general trend clearer. Even towards the end of the depicted steps, the loss does not appear to have fully converged to a flat plateau, suggesting that the model is still undergoing refinement on this batch of data. This pattern of rapid initial improvement followed by more gradual, albeit noisy, optimization was generally observed across the training of other master batches as well.

The complete set of loss progression graphs for all 13 master batches, detailing the training dynamics within each, are available. These graphs are organized into individual folders corresponding to each respective model version and can be accessed within the linked Kaggle dataset: [Finetuned Model Versions]. This allows for a more granular inspection of the model's learning process at each stage of the iterative fine-tuning.

In total, we trained our model on 169,659 images across 13 batches and tested it on a strategically curated test dataset of 34,000 images. The training process took approximately 2,730 minutes, while testing required around 3,120 minutes.

Evaluation

After the full iterative training cycle (or at intermediate stages after each master batch), the model was evaluated on a dedicated test dataset.

Evaluation Setup

- The processor and fine-tuned model were loaded from their saved location (i.e., `/kaggle/working/model_late`
- Hugging Face Accelerator was utilized for possibly distributed inference.
- An instance of `VQADataset` for the test set (e.g., from `/kaggle/input/master-test/test_dataset`) was created.
- A `DataLoader` was utilized to batch the test set.
- Mixed-precision inference (`autocast`) was employed for performance.

How Evaluation Works

1. **Initialization:**
   - The model goes into evaluation mode (`model.eval()`).
   - Gradient computation is turned off (`torch.no_grad()`) for performance savings.
   - Automatic Mixed Precision (`autocast`) is turned on if available, which saves memory and can speed up inference.
   - Evaluation state (processed batch indices, collected predictions, correct/total counts) is loaded from a `resume_file` if one exists. Otherwise, evaluation begins from scratch. Variables such as `initial_batches_processed`, `processed_indices`, `predicted_all`, `true_all`, `correct`, and `total` are initialized or filled based on the loaded state.
2. **Iteration with Resume:**
   - The code loops over the batches yielded by the `test_loader`.
   - A `tqdm` progress bar is used to report progress, set up to report the number of batches already executed (`initial_batches_processed`) from the resume state.
   - Within each `batch_idx`, a conditional check is performed: `if batch_idx < initial_batches_processe` If this is true, this batch was executed in a previous run.

- When a batch is recognized as already processed, `pbar.update()` is invoked to explicitly increment the progress bar, and `continue` bypasses the processing code of this batch.

3. **Batch Processing (for unprocessed batches):**

   - Input data (`pixel_values`, `input_ids`, `attention_mask`) is moved to the appropriate `device`.
   - The model's `generate` method is called with the input data to produce `generated_ids`. Generation parameters (`max_length`, `do_sample=False`, `num_beams=1`) control the output.
   - The `generated_ids` are decoded into human-readable strings (`predicted_answers`) using the `processor`.
   - Ground truth `answers` (labels) are also decoded into strings (`decoded_answers`).

4. **Accuracy Calculation and Storage:**

   - The code iterates through each sample within the current batch, comparing the predicted and true answers.
   - Answers are cleaned by stripping whitespace and converting to lowercase (`.strip().lower()`) for case-insensitive comparison.
   - The cleaned predicted and true answers are appended to `predicted_all` and `true_all` lists, respectively.
   - If the cleaned predicted answer exactly matches the cleaned true answer, the `correct` counter is incremented.
   - The `total` counter (representing total samples processed so far) is incremented.

5. **State Saving (Checkpointing):**

   - The current `batch_idx` is added to the `processed_indices` set.
   - The evaluation state is periodically saved to the specified `resume_file`. The saving condition triggers roughly every 1000 total samples processed (`total % 1000 < batch_size` and `total` $\geq$ `1000`) and explicitly at the very end of the evaluation loop (`batch_idx == batch_count - 1`).
   - The saved state includes the list of processed batch indices (`indices`), the accumulated predictions (`predicted`), true answers (`true`), and the current `correct` and `total` counts, stored as a JSON object.

Standard Metrics

Exact Match (EM)

- **Indication:** Measures the percentage of generated answers that are an exact, character-for-character match to one of the ground truth reference answers.
- **Justification:** Essential for evaluating questions with definitive, short answers where precision is paramount. It provides a clear measure of the model's ability to produce factually correct and precisely phrased responses for specific queries.

- **Limitation for one-word VQA:** While suitable for questions with a single, unambiguous one-word answer, it harshly penalizes minor variations (e.g., singular vs. plural if not specified, or an equally valid synonym not present in the ground truth). It offers no partial credit for semantically very close but not identical single-word answers.

Additional Metrics

BERTScore - Precision

- **Indication:** Quantifies how much of the generated answer is semantically similar to the reference answers, leveraging contextual embeddings from BERT. A higher score indicates that the model's output is relevant and avoids generating irrelevant information.
- **Justification:** Moves beyond simple word overlap to assess semantic equivalence. Crucial for VQA as valid answers can be phrased in multiple ways. It helps to penalize the inclusion of incorrect or unsubstantiated details in the generated response.
- **Limitation for one-word VQA:** For a single generated word, precision will largely indicate if that word is semantically aligned with the reference word(s). However, the complexity of BERT embeddings might be excessive for single-word comparisons and could potentially assign high precision to a semantically related but factually incorrect single word.

BERTScore - Recall

- **Indication:** Measures the extent to which the generated answer covers the information present in the reference answers, using BERT embeddings for semantic comparison. A higher score suggests the model is providing comprehensive answers that capture the key aspects of the ground truth.
- **Justification:** Important for evaluating responses to open-ended questions that may require describing multiple elements or aspects of the image. It assesses if the model is effectively extracting and presenting the relevant information from the visual context.
- **Limitation for one-word VQA:** If both generated and reference answers are single words, recall behaves very similarly to precision. If the model is expected to produce only one word, its ability to "cover" more information (which recall measures) is inherently limited, making this aspect less informative than for longer answers.

BERTScore - F1

- **Indication:** The harmonic mean of BERTScore Precision and Recall, offering a balanced measure of semantic similarity between the generated and reference answers.
- **Justification:** Provides a single, robust score that reflects the overall semantic overlap and relevance of the generated answer, accounting for both the precision of the generated content and the coverage of the reference information. Often considered a primary metric for semantic evaluation.
- **Limitation for one-word VQA:** Inherits limitations from BERTScore Precision and Recall for single words. While it balances semantic presence and relevance, it might still score a single, semantically similar but incorrect word highly. Its nuanced balancing act is more impactful for multi-word answers.

BARTScore

- **Indication:** Utilizes a pre-trained BART model to evaluate the quality of the generated text by assessing its likelihood within the BART language model, potentially capturing aspects like fluency, coherence, and factual consistency in a generation-aware manner.

- **Justification:** Offers a more advanced, model-based evaluation that goes beyond surface-level text matching. It can provide insights into the naturalness and quality of the generated language, which is important for user-facing VQA applications.
- **Limitation for one-word VQA:** Concepts like fluency and coherence are minimally applicable to single-word answers. BARTScore might favor common words over rare but correct single-word answers due to the language model's training distribution. Its strengths in evaluating the structure of generated text are underutilized for single tokens.

BLEU Score

- **Indication:** Measures the n-gram overlap between the generated answer and the reference answers, with a penalty for overly short generations. Primarily assesses the precision of word sequences.
- **Justification:** A traditional metric for evaluating text generation quality, particularly useful for assessing how well the model replicates common phrases and word combinations found in human-provided answers. While sensitive to exact phrasing, it offers a foundational measure of textual similarity.
- **Limitation for one-word VQA:** For single-word answers, BLEU (especially BLEU-1) essentially reduces to an exact match. Higher-order n-grams (BLEU-2, BLEU-3, BLEU-4) will typically be zero if the generated and reference answers are single words (unless they are identical), offering little discriminative power. The brevity penalty is also less relevant.

ROUGE-L

- **Indication:** Focuses on the Longest Common Subsequence (LCS) between the generated and reference answers, measuring the overlap in the longest shared sequence of words, irrespective of their order. Provides an F1-like score based on LCS.
- **Justification:** Relevant for evaluating answers where the order of words might vary but the presence of a significant common sequence indicates shared content. Useful for assessing if the model captures the main informational flow or key phrases from the reference.
- **Limitation for one-word VQA:** If both the generated and reference answers are single words, ROUGE-L effectively becomes a binary exact match (score 1 if identical, 0 otherwise). It cannot capture semantic similarity between different single words.

METEOR

- **Indication:** Calculates an alignment-based score considering exact word, stem, synonym, and paraphrase matches between the generated and reference answers. Designed to correlate better with human judgments than just n-gram overlap.
- **Justification:** Addresses the limitations of purely surface-level metrics by incorporating semantic equivalence through synonymy and paraphrasing. Provides a more human-like evaluation of answer correctness when there are variations in wording.
- **Limitation for one-word VQA:** While its ability to match synonyms is beneficial for single-word answers (making it more flexible than EM), the more complex aspects of METEOR like fragmentation and alignment penalties are less impactful for single words. It might give a good score to a synonym that is technically correct but not the most appropriate or common answer.

Jaccard Similarity

- **Indication:** Measures the ratio of the intersection to the union of the sets of unique tokens in the generated and reference answers. Indicates the degree of overlap in the vocabulary used.

- **Justification:** Offers a simple, set-based measure of token overlap. Useful for understanding the common words shared between the generated and ground truth answers, providing a basic indication of content overlap ignoring word order and frequency.

- **Limitation for one-word VQA:** For single-word answers, Jaccard Similarity becomes binary: it is 1 if the single generated word is identical to the single reference word, and 0 if they are different. It cannot distinguish between a completely unrelated word and a semantically close synonym.

Sørensen–Dice Coefficient

- **Indication:** Another set-based metric ($2 \times |A \cap B| / (|A| + |B|)$) quantifying the overlap between the sets of tokens in the generated and reference answers. Similar to Jaccard Similarity but can be less sensitive to the size of the sets.

- **Justification:** Provides an alternative measure of token overlap, reinforcing the analysis of shared vocabulary between the model's output and the reference answers.

- **Limitation for one-word VQA:** Similar to Jaccard Similarity, for single-word answers, this coefficient becomes 1 if the words are identical and 0 if they are different. It does not provide any partial credit for semantic similarity for non-identical single words.

LCS Ratio

- **Indication:** The ratio of the length of the Longest Common Subsequence (LCS) to the length of the reference answer. Indicates the proportion of the reference answer's word sequence captured by the generated answer.

- **Justification:** A straightforward metric focusing specifically on the extent to which the generated answer preserves the order and content of the longest common sequence of words from the reference, offering insight into sequential overlap.

- **Limitation for one-word VQA:** If the reference answer is a single word, the LCS ratio will be 1 for an exact match and 0 for any non-match. It offers no nuance for single words that might be semantically related but not identical.

Fuzzy Matching Score

- **Indication:** Measures the similarity between strings that may contain minor differences, such as typos or slight variations in spelling. Scores are based on the number of edits required to match the strings.

- **Justification:** Important for VQA evaluation to account for potential minor errors in the model's output that do not fundamentally alter the correctness or meaning of the answer. Prevents penalizing models for small textual imperfections.

- **Limitation for one-word VQA:** While useful for typos, it might incorrectly assign high similarity to single words that are orthographically close but semantically distinct (e.g., ”horse” vs. ”house”). For single-word answers, it's crucial that the ”fuzziness” doesn't obscure actual incorrectness beyond minor misspellings of the correct word.

VQA Accuracy

- **Indication:** A standard VQA-specific metric that considers a generated answer correct if at least 3 out of 10 human annotators provided that answer as a ground truth.

- **Justification:** Developed to handle the inherent subjectivity and variability in VQA answers. It provides a more realistic assessment of performance by acknowledging that multiple valid answers can exist for a single image-question pair.

- **Limitation for one-word VQA:** If the pool of human-annotated answers for a given question primarily consists of specific single words, this metric might not fully credit an equally valid, synonymous single-word answer if that synonym wasn't provided by at least three annotators. Its effectiveness for unique one-word answers depends heavily on the diversity and exhaustiveness of the collected ground truth answers.

Proposed Metrics

Visual-Contextual Consistency Score (VCCS)

- **Indication:** A metric designed to determine the degree to which the response generated is aligned not just with the question and answer text reference, but also with the visual content of the image. It should be designed to determine whether the entities, attributes, and relations stated in the answer do indeed exist and align with the image.
- **Explanation:** Very crucial for VQA as it quantifies how much a model is able to base its linguistic answer on the visual input. Good VCCS indicates the model is indeed "seeing" and drawing conclusions from the image to create its answer, and not merely relying on language priors. This is critical to build robust and stable VQA systems.

Token-Level Overlap

- **Indication:** Generates an overt token-level overlap breakdown between the reference and generated responses. This could include classifying similar overlapping tokens (e.g., by part of speech) or analyzing their distribution.
- **Rationale:** Offers a diagnostic instrument for understanding *what kind* of information the model is actually conveying or generating relative to the ground truth. Helps to identify specific strengths or weaknesses in the model's language generation relative to the visual input.

By using this comprehensive set of metrics, we aim to provide a thorough and in-Depth evaluation of the base and fine-tuned BLIP models, highlighting their performance across various dimensions of VQA.

Metric	**Score**
Exact Match (EM)	13.28%
BERTScore - Precision	0.9487
BERTScore - Recall	0.9310
BERTScore - F1	0.9388
BLEU Score	0.0248
ROUGE-L	0.1418
METEOR	0.0853
Jaccard Similarity	0.1369
Sørensen–Dice Coefficient	0.1387
LCS Ratio	0.1370
Token-Level Overlap	0.1369
Fuzzy Matching Score	0.2818
VQA Accuracy	13.28%
Visual-Contextual Consistency Score (VCCS)	0.1373
BARTScore	-6.3153

Table 4: Baseline BLIP model results across 15 metrics

Metric	**Score**
Exact Match (EM)	20.53%
BERTScore - Precision	0.9564
BERTScore - Recall	0.9383
BERTScore - F1	0.9464
BLEU Score	0.0374
ROUGE-L	0.2143
METEOR	0.1105
Jaccard Similarity	0.2091
Sørensen–Dice Coefficient	0.2109
LCS Ratio	0.2093
Token-Level Overlap	0.2091
Fuzzy Matching Score	0.3840
VQA Accuracy	20.53%
Visual-Contextual Consistency Score (VCCS)	0.2094
BARTScore	-5.8659

Table 5: Fine-tuned BLIP model (v13) results across 15 metrics

5 BLIP Model Fine-tuning for Visual Question Answering (VQA) Performance Analysis

5.1 Detailed Results

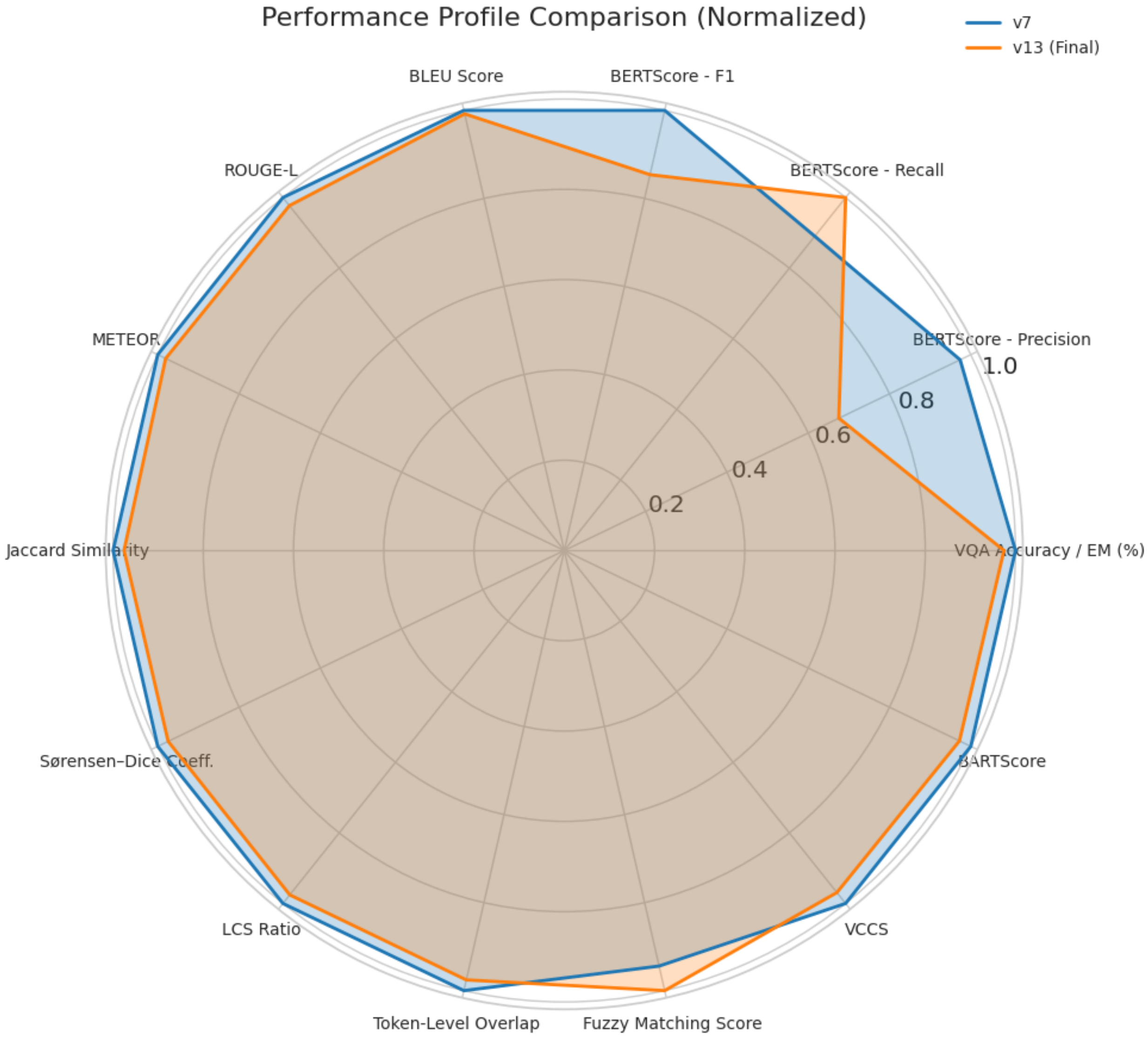


Figure: Performance Comparison of our 2 Best Models (V-7 and V-13)

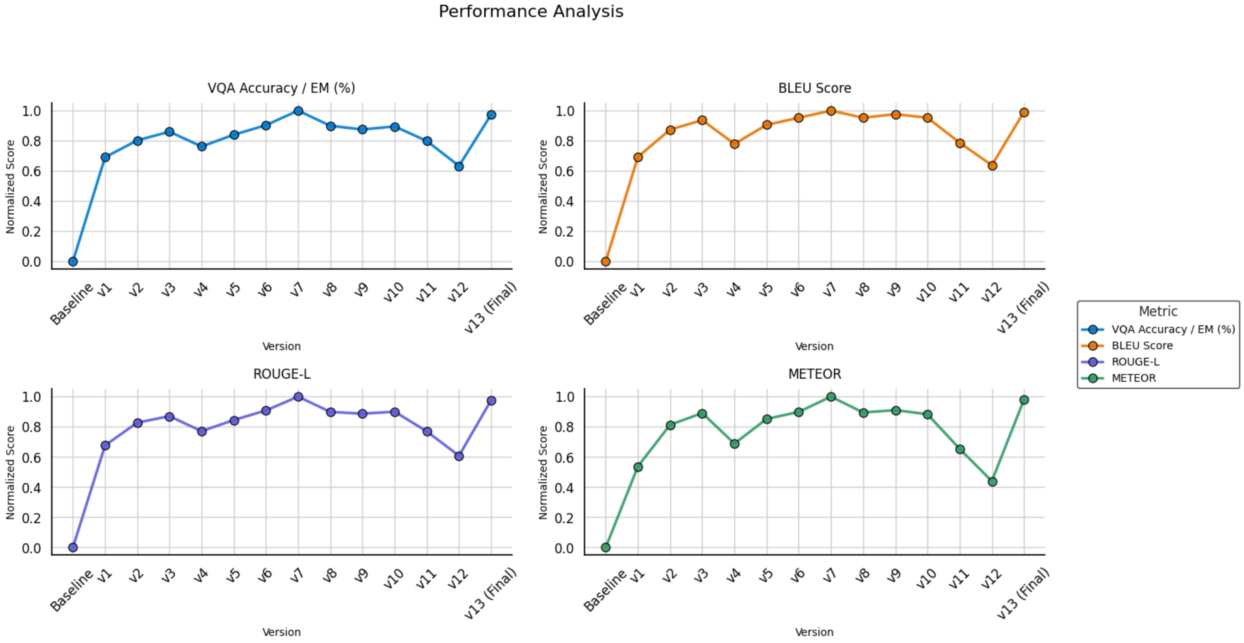


Figure 15: Performance Comparison

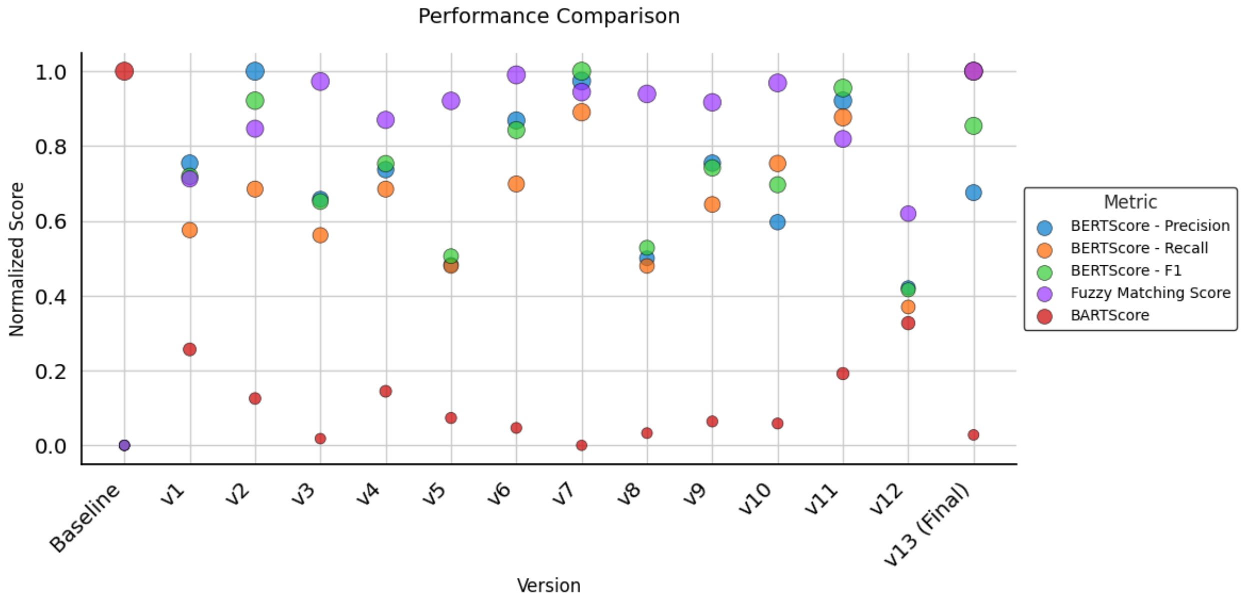


Figure 16: Performance Comparison

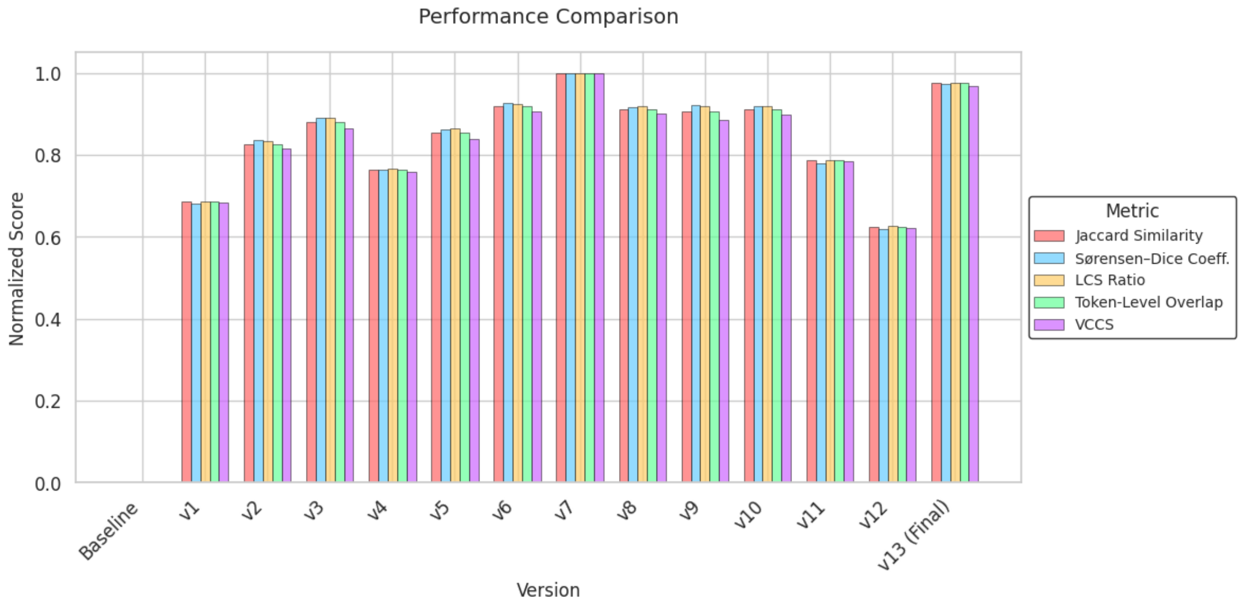


Figure 17: Performance Comparison

The complete evaluation metric decomposition for all 13 fine-tuned versions and the baseline can be found in the directory: `/EvaluationMetrics/Results`.

This section introduces the performance testing of the BLIP model fine-tuned through 13 iterative steps. Fine-tuning enhanced the model's ability to comprehend visual input and generate valid text answers to image-based questions. The results show significant overall improvement in VQA accuracy compared to the baseline, with an initial rapid improvement, peak performance around version 7, a decline near versions 11 and 12, and a strong recovery by version 13 across most evaluation metrics.

5.2 Overall Trend

The fine-tuning process improved the model's visual question answering capabilities. Performance metrics rose well above the baseline, peaked at version 7, declined in versions 11 and 12, and rebounded near peak levels in version 13.

5.3 Key Metrics Their Trends

Performance was evaluated using a variety of metrics to assess the model's capabilities:

- **VQA Accuracy / Exact Match (EM):**
  - Increased sharply from Baseline (13.28%) to v1 (18.42%).
  - Consistent improvement up to v7 peak (20.72%).
  - Declined during v8-v10, dropped further in v11 (19.22%) and v12 (17.99%).
  - Rebounded strongly in v13 (20.53%), nearly matching v7.
- **Text Generation Metrics (BERTScore F1, ROUGE-L, METEOR, BLEU, Jaccard, Sørensen–Dice, LCS Ratio, Token-Level Overlap, Fuzzy Matching):**
  - Mirrored the VQA Accuracy trend with big gains from baseline to v1, peaking at v7.
  - Declined in v11/v12, then recovered in v13.
  - BERTScore F1 remained stable and high (0.9388 baseline to 0.9477 peak).
  - Fuzzy Matching peaked slightly differently at v3 (0.3812) and v13 (0.3840).
- **Visual-Contextual Consistency Score (VCCS):**
  - VQA-specific metric assessing answer-image consistency.
  - Increased from Baseline (0.1373) to v7 peak (0.2117), dropped at v12 (0.1836), then rose again at v13 (0.2094).
- **BARTScore:**
  - Quality score (lower is better).
  - Improved from Baseline (-6.3153) to v7 peak (-5.8530).
  - Followed overall performance fluctuations with reduction at v11/v12.
  - Strong recovery at v13 (-5.8659).

5.4 Final Takeaways

- The largest single improvement happened during initial fine-tuning (Baseline to v1).
- Version 7 is the top-performing model, with the highest VQA Accuracy and peak values in other metrics.
- Versions 11 and 12 experienced noticeable performance drops.
- Version 13 recovered strongly, nearly matching the top performance of v7, and significantly exceeding the baseline.

In conclusion, fine-tuning successfully enhanced the BLIP model's VQA performance. Despite fluctuations, the overall trend across the 13 versions is positive, with final version 13 showing robust performance improvements across all key metrics.

6 Experiments Conducted

6.1 During Dataset Generation and evaluations

Two methods for generating VQA datasets were evaluated: one utilizing Gemini and another using Llama. Initial testing revealed that the dataset generated by Gemini led to superior performance in downstream VQA tasks compared to the dataset generated by Llama. Consequently, the Gemini-generated dataset was selected for subsequent fine-tuning experiments.

6.2 During Model Selection and Fine-tuning

The experiments considered fine-tuning with several BLIP model variants:

6.2.1

Considering BLIP-1

- **BLIP Base**: $\approx$ 214M parameters. Pretrained on image-text pairs; uses ViT-B encoder and a transformer decoder.
- **BLIP Large**: $\approx$ 336M parameters. Larger ViT-L encoder with more capacity and accuracy.

6.2.2

Considering BLIP-2 BLIP-2 separates the vision encoder and a frozen large language model (LLM), bridging them with a Querying Transformer (Q-Former).

- **BLIP-vqa-base**: 350M parameters. This variant was chosen for fine-tuning.
- **BLIP-2 with OPT-2.7B**: $\approx$ 3B parameters. ViT-G with OPT-2.7B.
- **BLIP-2 with FLAN-T5 XL**: $\approx$ 3B parameters. ViT-G with FLAN-T5 XL (3B).
- **BLIP-2 with FLAN-T5 XXL**: $\approx$ 11B parameters. ViT-G with FLAN-T5 XXL (11B).

Considering the trade-off between training time, available resources, and expected accuracy, the **BLIP-vqa-base** model was selected as the primary model for fine-tuning.

6.3 Data Sampling Strategies

We actually have taken into account the impact of sampling algorithms on our model Performance,by comparing 2 approaches to create train test splits from the generated dataset.

- **Random Sampling**: Data points were split randomly into training and testing sets.
- **Proportion-based Tired Sampling Algorithm**: This algorithm aimed to create splits that maintained the original distribution of certain characteristics within the dataset.

Experimental results demonstrated that the dataset generated using the proportion-based tired sampling algorithm led to better model performance compared to random sampling.

6.4 Evaluation and Performance

The fine-tuned BLIP-vqa-base model, trained on the dataset sampled using the proportion-based tired sampling algorithm, was evaluated on a randomly sampled test set of 10,000 data points. The evaluation yielded an exact match (EM) accuracy of approximately 40%.

6.5 Detailed Experimental Description of LoRA Fine-tuning Parameters

Low-Rank Adaptation (LoRA) was employed to fine-tune the selected BLIP-vqa-base model efficiently. LoRA injects small, trainable low-rank matrices into specific layers of the pre-trained model. The change in the weight matrix ΔW for a layer is approximated as the product of two lower-rank matrices, $\Delta W = AB$, where $A \in R^{d\times r}$ and $B \in R^{r\times k}$, and r is the LoRA rank ($r \ll \min(d, k)$).

Key parameters used during LoRA fine-tuning include:

- `r` (LoRA Rank): Defines the dimensionality of the low-rank matrices. A higher value increases trainable parameters and potential adaptation capacity but also resource requirements.
- `lora_alpha` (Scaling Factor): Scales the learned LoRA weights, with the scaling factor typically being $\frac{\alpha}{r}$. Influences the degree of adaptation.
- `lora_dropout` (Dropout): Applies dropout to the LoRA layers for regularization to prevent overfitting.
- `target_modules` (Target Modules): Specifies the layers within the pre-trained model where LoRA matrices are injected (e.g., attention layers like `q_proj`, `v_proj`).
- **Learning Rate**: Controls the step size for updating the LoRA parameters during optimization.
- **Number of Epochs**: The number of passes over the entire training dataset.
- **Batch Size**: The number of samples processed in each training iteration.
- **Optimizer**: The algorithm used for updating the LoRA parameters.

Determining the optimal values for these LoRA parameters was a significant undertaking in our fine-tuning process. We navigated a complex hyperparameter landscape where the interplay between settings, such as LoRA rank `r`, `lora_alpha`, and the learning rate, heavily influenced outcomes. Each experimental run to test a particular configuration was computationally intensive, especially considering our iterative training strategy across multiple master batches. This placed practical limits on the breadth of systematic explorations, such as exhaustive grid searches. Our tuning strategy, therefore, primarily involved an iterative approach. We initiated our experiments with baseline values informed by common practices in LoRA applications and findings from related literature. From this starting point, we proceeded with a series of targeted manual adjustments, carefully observing changes in validation metrics after each modification. The central challenge throughout this phase was to strike a balance between enhancing model performance on our specific VQA task, managing the available computational resources effectively, and ensuring the model generalized well without overfitting. This iterative refinement process ultimately guided us to the specific parameter set employed for the results presented in this report.

7 Challenges Faced

7.1 During Dataset Generation and evaluations

- The primary bottleneck of our entire process was dataset preparation, without which we couldn't move forward. Therefore, we had to devise a new method to create the dataset.
- While passing the metadata, the `"bullet point"` field often contained irrelevant information such as product dimensions and weight — details that could not be inferred from the image. To address this, we had to explicitly clarify this limitation in the prompt.
- The API key requirements were critical: a single API key could generate only 15 Q-A pairs for 750 images, which posed constraints we had to carefully manage.
- Initially, the quality of the generated questions was subpar. However, through continuous prompt refinement, we finally achieved a state-of-the-art data set.
- We were a bit confused in the initial stages wether to choose llama or gemeni, but by manual checking of the quality of q and a pairs,we finally chose gemini. which consumed a lot of time.

7.2 During Model Selections

- There were pretty good models,giving quite good accuracy, for instance, blip-2 3B while almost touching an exact match of 30 percent on our curated test-set.and 50 percent on the Blip-2 11 b params
- We also enquired about Bakllava,qwen models, even they were giving an accuracy of 20-30 percent on our curated dataset.
- But on the other hand blip-vqa-base 350M a very light model was only giving an exact match of almost 14 percent,But considering the resources available,time Requirements and dataset size, the estimated time of training for blip-vqa-base was prettly less as compared to the other heavier model.also we thought the Exact match wouldn't increase quite good in heavier models as they were already good.

7.3 During Training

- The whole finetuning process took us almost 7 days, as it was not a prallel process rather a sequential one, as we had to pass on the finetuned model to others once done,therby finetuning on the finetuned model
- In In the initial stages when we checked the accuracy of the fine-tuned model (v-1) and (v-2),we observed a steep decline in accuracy, which was mainly due to the inconvenience of

storing config.json after each fine-tuning process.

- We initially were only using 30-40 percent of the gpu, leading to an estimated time of 7-8 h for training, and after increasing the batch size to 64 or even higher.100 the estimated time dropped to 4 h.
- Using NVIDIA Tesla P100 GPUs on Kaggle poses challenges like limited 16 GB VRAM, which restricted training our models or large batch sizes. The older Pascal architecture of the P100 lacks some modern features, such as advanced tensor cores for efficient mixed precision training, resulting in slower performance compared to newer GPUs. Additionally, high demand on Kaggle can lead to longer queue times and limited availability.
- For the Tesla T4, although it also has 16 GB VRAM, its lower FP32 compute power means slower training for some workloads. It's optimized for mixed-precision but adapting code for this can be tricky. T4s may also experience thermal throttling under sustained heavy use, affecting performance stability. On Kaggle, both GPUs face common limits like weekly usage caps, shared resource variability, and fixed software environments that can constrain flexibility

7.4 During Inference

- Storing both predictions and ground truths for large datasets can consume significant disk and memory, especially with high-resolution outputs (e.g., image masks, long texts, large arrays). This issue worsens with multiple model variants or checkpoints, and formats like JSON or CSV add overhead, causing slow I/O and versioning challenges.
- Saving predicted and ground truth outputs for large datasets is storage-intensive, particularly with detailed outputs. When multiple models or checkpoints are involved, and formats like JSON or NumPy are used, redundancy and metadata inflate storage needs, leading to slower I/O and complex result management.
- Metrics like BARTScore and LCS are slow on large datasets. BARTScore requires running transformer models per prediction-reference pair, while LCS is computationally intensive for long sequences—both causing delays in the evaluation pipeline, especially without GPU support.
- Graphically representing all 15 metrics across 14 models (baseline + 13 versions) was challenging due to visual clutter and interpretability issues. To manage this, we divided the metrics into three categories—lexical, semantic, and structural—and created separate plots for each, allowing for clearer comparisons and better insights

Additional contribution/novelty

During Dataset Generation and evaluations

- Uninterrupted 24/7 script execution.
- Efficiently managed and executed multiple scripts concurrently on a single machine, optimizing resource usage.
- Performed comprehensive data preprocessing, including dataset exploration, integration, and refinement to ensure data quality and consistency.

- Proportional Tiered Sampling Algorithm.
- Prompt Engineering for VQA Generation.

During Training

- **Novel Iterative LoRA Fine-tuning Strategy:** Developed and successfully executed a novel iterative LoRA fine-tuning strategy, segmenting the extensive dataset into 14 'master batches'. This allowed for manageable, sequential training stages, effectively overcoming memory and computational limitations typically prohibitive for direct full-dataset fine-tuning with available resources.
- **Robust Multi-Stage Training Management:** Engineered a resilient training pipeline featuring automated checkpointing of model weights, LoRA adapter configurations, processor states, optimizer states, and loss history at the end of processing each master batch. This system, integrated with versioned Kaggle Datasets (e.g., `model_latest_vN`), ensured fault tolerance and seamless continuation for progressive learning across all stages.
- **Resource-Conscious LoRA Configuration and Optimization:** Methodically tuned LoRA-specific hyperparameters (such as rank `r`, `lora_alpha`, and `target_modules`) specifically for the BLIP model architecture through iterative experimentation. This process focused on achieving a practical balance between the model's adaptation capacity for our VQA task and the efficient utilization of limited GPU resources throughout the prolonged, multi-stage training period.
- **Dynamic Data Handling for Staged Learning:** Employed our custom `VQADataset` PyTorch class within the iterative training loop for efficient, on-the-fly data loading, consistent preprocessing, and correct multimodal input formatting. This was crucial for reliably handling diverse data segments as the model progressed through the multiple fine-tuning stages.

During Inference

- Came up with two new Metrics: VCCS and Token-Level-Overlap, which give in-depth understanding of the results.