**XrayGPT**

**1. Data Preprocessing/Loading (MIMIC-CXR)**

**Paper:**

* Removal of incomplete reports lacking finding or impression sections.
* Elimination of reports that have finding sections containing less than 10 words.
* Exclusion of reports with impression sections containing less than 2 words.
* Clean summaries are generated using GPT-3.5-turbo.
* Data is organized as images + filtered captions.

**Code:**

* Preprocessing: SeeREADME-DATASET.md for detailed steps.
* Processed data:

dataset/mimic/image/(images),dataset/mimic/filter\_cap.json (annotations).

* Loading:
  + [xraygpt/datasets/datasets/mimic\_dataset.py](https://github.com/gontamar/XrayGPT/blob/main/xraygpt/datasets/datasets/mimic_dataset.py) (MIMIC Dataset)
  + Loads images and captions from the dataset folder.
  + Builder:xraygpt/datasets/builders/image\_text\_pair\_builder.py (uses MIMIC Dataset and annotation paths).

**Preprocessing Steps**

### 1. **Removal of incomplete/short reports**

* Remove reports without both findings and impression; findings < 10 words; impression < 2 words.

**Where:**

* This logic is implemented in the data preprocessing scripts provided in the repo, referenced in [README-DATASET.md](https://github.com/gontamar/XrayGPT/blob/main/README-DATASET.md).
* **Script location:**
* You will usually find this in a custom Python preprocessing script (often not tracked in the main repo, but referenced in the dataset README).
* The output of this step is the filter\_cap.json file.

### 2. **Cleaning of text (removal of prior comparisons, de-identified symbols, and view info)**

* Remove sentences with prior history, "\_\_" symbols, view info.

**Where:**

* Also part of the preprocessing scripts referenced in [README-DATASET.md](https://github.com/gontamar/XrayGPT/blob/main/README-DATASET.md).

**Code pattern:**

* Regular expressions or string matching to drop sentences with phrases like "compared to prior", symbols "\_\_", and view description lines ("PA and lateral views", etc.).
* The code for this is typically in a Jupyter notebook or a Python script used before training; sometimes these scripts are not included in the repo but their logic is described in the README.

### 3. **Summarization with GPT-3.5-turbo**

* Use GPT-3.5-turbo to create a single, high-quality summary from cleaned findings/impression.
* This is described as an external step in the documentation, not something automated in the training code itself.

**How:**

* The cleaned findings/impression are fed to GPT-3.5-turbo via OpenAI API or batch script, and the result is stored as the summary in the JSON annotation file.
* **Output:**
* The result is included in the final filter\_cap.json annotation file for each image.

### 4. **Pairing with images**

* Pair processed summary with the corresponding image using unique identifiers.

**Where:**

* Assembling these pairs is part of the preprocessing script, outputting filter\_cap.json (a list of image paths and their summaries).

### 5. **Final dataset loading**

* Dataset contains only high-quality, cleaned, summarized image-caption pairs.
* **Where (in code):**
* **File:** [xraygpt/datasets/datasets/mimic\_dataset.py](https://github.com/gontamar/XrayGPT/blob/main/xraygpt/datasets/datasets/mimic_dataset.py)
* **Class:** MIMICDataset
* Loads each image and its paired summary from filter\_cap.json.
* **Builder:** [xraygpt/datasets/builders image\_text\_pair\_builder.py](https://github.com/gontamar/XrayGPT/blob/main/xraygpt/datasets/builders/image_text_pair_builder.py)
* Assembles the final dataset object for training/evaluation.

**2. Model Architecture (Vision Encoder, Language Model, Fusion)**

**Paper:**

* Vision encoder: Frozen MedCLIP/EVA ViT.
* Language model: Vicuna (LLaMA-based).
* Q-Former for vision-language fusion.

**Code:**

* Vision encoder: xraygpt/models/eva\_vit.py (EVA ViT).
* Q-Former: xraygpt/models/Qformer.py (BERT-style transformer for modality bridging).
* Language model: xraygpt/models/modeling\_llama.py (LLaMA/Vicuna).
* Integration: xraygpt/models/mini\_gpt4.py (multimodal pipeline).

**3. Training Scripts/Configuration (Hyperparameters, Loss Functions)**

**Paper:**

* Two-stage: pretrain on MIMIC, finetune on Open-i.
* Distributed training, Adam optimizer, learning rate scheduling.

**Code:**

* Main script: train.py ( all training stages).
* Config: YAML files in train\_configs/, parsed by
* ( xraygpt/common/config.py.)
* Optimizer: Adam, created in ( xraygpt/runners/runner\_base.py.)
* Scheduler: xraygpt/common/optims.py (e.g.,LinearWarmupCosineLRScheduler).
* Loss functions: Defined within model/task classes (see xraygpt/models/ and xraygpt/tasks/).

**Steps for training flow**

## **1. Launching Training**

*python train.py --cfg-path <config.yaml>*

The config file specifies:

* Which dataset to use and where it is located (e.g., images directory, CSV, etc.)
* Model architecture and hyperparameters
* Task type (classification, report generation, etc.)
* Training parameters (epochs, batch size, optimizer, etc.)

2. Argument Parsing and Config Loading

* **File:** train.py → xraygpt/common/config.py
  + Parses CLI arguments for the config file path.
  + Loads the YAML/JSON config into a structured object (cfg).
  + This config centrally controls all subsequent steps.

**3. Distributed Setup, Seeding, and Logging**

* **Files:**
  + Distributed: *xraygpt/common/dist\_utils.py*
  + Seeding: *train.py*
  + Logging: *xraygpt/common/logger.py*
* **What Happens:**
  + Sets up distributed training if required (e.g., multi-GPU).
  + Sets random seeds (Python, NumPy, PyTorch) for reproducibility.
  + Initializes logging (so only the main process logs messages).

4. Task and Dataset Preparation

**A. Task Selection**

* **File:** xraygpt/tasks/\_\_init\_\_.py and relevant task module
  + *Calls setup\_task(cfg)*, which instantiates a task class (e.g., Classification, Report Generation).
  + The Task object knows how to build the right datasets and models for this task.

### **B. Dataset Loading and Processing**

* **Files:**
  + xraygpt/datasets/builders/ (the dataset builder for your data, e.g., MIMICBuilder.py)
  + xraygpt/processors/ (for transforms and preprocessing)
  + **Task.build\_datasets(cfg)**
    - Reads the config to determine which dataset builder to use.
    - Calls the appropriate builder (e.g., MIMICBuilder for MIMIC-CXR dataset).
  + **Inside the Builder:**
    - Loads raw data from disk (images, labels, etc.).
    - Applies transformations (preprocessing/augmentation) using processor classes specified in config, such as:
      * Resizing, normalization, cropping for images (xraygpt/processors/blip\_processors.py)
      * Text cleaning/tokenization for reports

(README-DATASET.md for tokenization)

* + - Splits data into train/val/test as configured.
    - Returns PyTorch Dataset objects (or dict with splits).
  + **Dataloader Construction (usually in Runner):**
    - These datasets are wrapped in PyTorch DataLoader objects for batching and shuffling data during training.

5. Model Construction

* **Files:** xraygpt/models/ (specific model architecture)
* **What Happens:**
  + ***Task.build\_model(cfg)*** reads model parameters from config.
  + Instantiates the model class (e.g., a Transformer for images/reports).
  + The model is registered in the global registry, allowing dynamic loading.

6. Runner Construction and Training Loop

* **Files:** xraygpt/runners/ (e.g., runner\_base.py)
  + xraygpt/common/optims.py (for optimizer and learning rate scheduler)
  + The runner is chosen based on config and registry (e.g., epoch-based runner).
  + The runner receives: config, job\_id, task, model, datasets.

### **Inside runner.train(): The Training Loop**

1. **Dataloader Setup:**
   * Wraps the dataset (from builder) in a DataLoader for batching.
2. **Optimizer and Scheduler Setup:**
   * Sets up optimizer (e.g., Adam, SGD) and learning rate scheduler as specified in config.
3. **Epoch Loop:**
   * For each epoch (number from config):
     + For each batch in the training dataloader:
       1. **Batch Fetching:**
          - Loads a batch of (images, labels, etc.) from the DataLoader.
       2. **Data Processing:**
          - Applies any additional on-the-fly transforms (if required).
       3. **Model Forward Pass:**
          - Passes the batch through the model to get predictions.
       4. **Loss Computation:**
          - Computes the loss function (e.g., cross-entropy) comparing predictions and targets.
       5. **Backward Pass:**
          - Computes gradients via loss.backward().
       6. **Optimizer Step:**
          - Updates model weights via optimizer.step().
       7. **Scheduler Step:**
          - Adjusts learning rate if using a scheduler.
       8. **Logging:**
          - Logs metrics (loss, accuracy, etc.).
     + **Validation/Evaluation:**
       1. At the end of each epoch (or at intervals): evaluates on the validation set.
     + **Checkpointing:**
       1. Saves model checkpoints periodically.

**4. Evaluation/Metrics**

**Paper:**

* Reports retrieval, similarity metrics, language quality.

**Code:**

* Evaluation loop: xraygpt/runners/runner\_base.py (eval\_epoch, evaluate).
* Metric computation: xraygpt/models/blip2.py (similarity matrices, recall, etc.).
* Logging and checkpointing: in runner\_base.py.