Preparing data and pre-processing

```
import torch
print(f"CUDA available: {torch.cuda.is_available()}")
if torch.cuda.is_available():
    print(f"GPU device: {torch.cuda.get_device_name(0)}")

# Install necessary packages
!pip install -q transformers datasets peft accelerate bitsandbytes evaluate rouge_score nltk bert_score trl scikit-lea
!pip install --upgrade transformers accelerate>=0.25.0
```

```
CUDA available: True
GPU device: NVIDIA A100-SXM4-40GB
```

Loading the dataset, Using Hugging Face api for authentication and loading the `PubMedQA` dataset and printing the dataset structure and an example data point to verify successful loading and understand the data format.

```
from huggingface_hub import login
from datasets import load_dataset
import pandas as pd
import random
import nltk

try:
    login("")
except Exception as e:
    print(f"{e}")

# Download necessary NLTK data
try:
    nltk.data.find('tokenizers/punkt')
except LookupError:
    nltk.download('punkt')

# Load the PubMedQA dataset
print("Loading PubMedQA dataset...")
dataset = load_dataset("qiaojin/PubMedQA", "pqa_labeled")
print("Dataset loaded successfully!")

print("\nDataset structure:")
print(dataset)

full_training_data = dataset["train"] # full 1000 examples
print(f"\nTotal number of labeled examples to be used: {len(full_training_data)}")

print("\nExample data point from full data:")
if len(full_training_data) > 0:
    example = full_training_data[0]
    for key, value in example.items():
        if key == 'context' and isinstance(value, dict) and 'contexts' in value:
             print(f"context['contexts'][0]: {value['contexts'][0][:150]}...") # Show start of first context
        elif isinstance(value, str) and len(value) > 100:
            print(f"{key}: {value[:100]}...")
        else:
            print(f"{key}: {value}")
else:
    print("Dataset is empty.")
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
Loading PubMedQA dataset...
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://h
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models
  warnings.warn(
README.md: 100% 5.19k/5.19k [00:00<00:00, 578kB/s]
train-00000-of-00001.parquet: 100% 1.08M/1.08M [00:00<00:00, 12.6MB/s]
Generating train split: 100% 1000/1000 [00:00<00:00, 19228.37 examples/s]

Dataset loaded successfully!

Dataset structure:
DatasetDict({
    train: Dataset({
        features: ['pubid', 'question', 'context', 'long_answer', 'final_decision'],
        num_rows: 1000
    })
})

Total number of labeled examples to be used: 1000

Example data point from full data:
pubid: 21645374
question: Do mitochondria play a role in remodelling lace plant leaves during programmed
context['contexts'][0]: Programmed cell death (PCD) is the regulated death of cells withi
long_answer: Results depicted mitochondrial dynamics in vivo as PCD progresses within the
final_decision: yes
```

This cell splits the loaded `PubMedQA` dataset into two subsets: a training set and a test set. Here, 10% of the data is reserved for the test set, and the remaining 90% is used for training.

```
from datasets import DatasetDict

if 'full_training_data' not in globals():
     raise NameError("Variable 'full_training_data' not found. Make sure the previous cell ran correctly.")

# 10% for test
split_percentage = 0.1
split_datasets = full_training_data.train_test_split(test_size=split_percentage, seed=42)

train_dataset_raw = split_datasets['train'] #  900 examples
test_dataset_raw = split_datasets['test'] # 100 examples

print(f"Raw Training set size: {len(train_dataset_raw)}")
print(f"Raw Test set size: {len(test_dataset_raw)}")
```

```
Raw Training set size: 900
Raw Test set size: 100
```

Defines a `SYSTEM_PROMPT` containing the system message, the question, and the context, along with the expected response format.

expected response format

```
# prompt for the medical assistant
SYSTEM_PROMPT = """You are a medical assistant trained to answer health related questions based on medical Data.
Use the provided context from the dataset to give accurate, helpful responses.
Base your answers only on the information provided in the context, and if you're unsure, acknowledge the limitations o

# Function to format examples for instruction fine-tuning
def format_instruction(example):
    question = example.get("question", "")
    context_dict = example.get("context", {})
    contexts = context_dict.get("contexts", [])
    full_context = " ".join(contexts) if contexts else ""
    answer = example.get("long_answer", "")
    final_decision = example.get("final_decision", "unknown")

    # Create the instruction format
    instruction = f"{SYSTEM_PROMPT}\n\nQuestion: {question}\n\nContext: {full_context}"

    # Format the response (target for the model to learn)
    response = f"{answer} The direct answer to your question is: {final_decision}."

    return {
        "instruction": instruction,
        "input": "",
        "output": response
    }

print("SYSTEM_PROMPT and format_instruction function defined.")
```

```
SYSTEM_PROMPT and format_instruction function defined.
```

```
print("Formatting the TRAINING dataset for fine-tuning...")

if 'train_dataset_raw' not in globals():
     raise NameError("Variable 'train_dataset_raw' not found. Make sure the data splitting cell ran.")
if 'format_instruction' not in globals():
     raise NameError("Function 'format_instruction' not found. Make sure the function definition cell ran.")

# formatting ONLY to the training split
try:
    formatted_train_dataset = train_dataset_raw.map(format_instruction)
except Exception as e:
     print(f" {e}")
     raise

print("\nExample of formatted TRAINING data:")
if len(formatted_train_dataset) > 0:
    example_train = formatted_train_dataset[0]

    instruction_text = example_train.get('instruction', '[Instruction missing]')
    input_text = example_train.get('input', '[Input missing]')
    output_text = example_train.get('output', '[Output missing]')

    print(f"INSTRUCTION (first 300 chars):\n{instruction_text[:300]}...\n")
    print(f"INPUT:\n{input_text}\n") #EMPTY
    print(f"OUTPUT (first 200 chars):\n{output_text[:200]}...\n")
else:
    print("Formatted training dataset is empty.")

print("Training data formatting complete.")
```

```
Formatting the TRAINING dataset for fine-tuning...
Map: 100%  900/900 [00:00<00:00, 4513.94 examples/s]
```

```
Map: 100% 900/900 [00:00<00:00, 4515.94 examples/s]

Example of formatted TRAINING data:
INSTRUCTION (first 300 chars):
You are a medical assistant trained to answer health related questions based on medical D
Use the provided context from the dataset to give accurate, helpful responses.
Base your answers only on the information provided in the context, and if you're unsure,

INPUT:


OUTPUT (first 200 chars):
Golytely was more efficacious than MiraLAX in bowel cleansing, and was independently assc

Training data formatting complete.
```

Baseline Model Evaluation

Establishing a performance baseline by evaluating the *original*, pre-trained Mistral 7B model on the test set *before* any fine-tuning.

```
import numpy as np
import pandas as pd
import evaluate
from tqdm.auto import tqdm
import re
import torch
from sklearn.metrics import f1_score, accuracy_score
import warnings
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
import gc

print("\n--- Phase 1: Baseline Model Evaluation ---")
warnings.filterwarnings("ignore", category=UserWarning, module='sklearn')


if 'test_dataset_raw' not in globals():
     raise NameError("Variable 'test_dataset_raw' not found. Make sure the data splitting cell ran.")
BASELINE_EVAL_SAMPLES = len(test_dataset_raw)
print(f"Will evaluate baseline on {BASELINE_EVAL_SAMPLES} test samples.")

print("Loading resources for baseline evaluation...")
try:
    # Load metrics
    rouge_metric = evaluate.load("rouge")
    bertscore_metric = evaluate.load("bertscore")

    base_model_id = "mistralai/Mistral-7B-v0.1"
    bnb_config_baseline = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.float16,
        bnb_4bit_use_double_quant=True
    )
    print(f"Loading base model: {base_model_id}")

    if 'model' in globals(): del model; gc.collect(); torch.cuda.empty_cache()
    if 'base_model' in globals(): del base_model; gc.collect(); torch.cuda.empty_cache()
    if 'ft_model' in globals(): del ft_model; gc.collect(); torch.cuda.empty_cache()
```

```
    if 'ft_model' in globals(): del ft_model; gc.collect(); torch.cuda.empty_cache()
    if 'base_model_ft' in globals(): del base_model_ft; gc.collect(); torch.cuda.empty_cache()


    base_model = AutoModelForCausalLM.from_pretrained(
        base_model_id,
        quantization_config=bnb_config_baseline,
        device_map="auto", # Automatically map layers
        trust_remote_code=True
    )
    base_tokenizer = AutoTokenizer.from_pretrained(base_model_id)

    if base_tokenizer.pad_token is None:
        base_tokenizer.pad_token = base_tokenizer.eos_token
    base_tokenizer.padding_side = "left"
    print("Base model and tokenizer loaded.")

except Exception as e:
    print(f"Error loading resources for baseline: {e}")
    raise

if 'SYSTEM_PROMPT' not in globals():
     SYSTEM_PROMPT = """You are a medical assistant trained to answer health related questions based on medical Data.
Use the provided context from the dataset to give accurate, helpful responses.
Base your answers only on the information provided in the context, and if you're unsure, acknowledge the limitations o
     print("Redefined SYSTEM_PROMPT for baseline generation.")


def extract_response_baseline(text):
    """Extracts text potentially following an [/INST] tag, cleaning special tokens."""
    text = text.strip()
    inst_end_pos = text.rfind("[/INST]")
    if inst_end_pos != -1:
        response = text[inst_end_pos + len("[/INST]"):].strip()
    else: response = text # Fallback
    response = response.replace("<s>", "").replace("</s>", "").strip()
    return response

def extract_decision_baseline(text):
    """Extracts 'yes'/'no'/'maybe' decision, tailored for base model output."""
    if not text or not text.strip(): return "unknown"
    text_lower = text.lower()
    first_part = text_lower[:250] # Check a reasonable chunk at the start
    if "is yes" in first_part or "answer: yes" in first_part or "final decision: yes" in first_part: return "yes"
    if "is no" in first_part or "answer: no" in first_part or "final decision: no" in first_part: return "no"
    if "is maybe" in first_part or "answer: maybe" in first_part or "final decision: maybe" in first_part: return "may
    # Simpler keyword check as fallback
    if "yes" in first_part: return "yes"
    if "no" in first_part: return "no"
    if "may depend" in first_part or "possibly" in first_part or "unclear" in first_part or "uncertain" in first_part:
    return "unknown" # Default

def generate_for_baseline(question, context=None, model_to_use=None, tokenizer_to_use=None):
    """Generates response using the baseline model."""
    if model_to_use is None or tokenizer_to_use is None: raise ValueError("Model/Tokenizer needed.")
    # Simple instruction prompt for base model, less specific than fine-tuning prompt
    # Using a slightly different prompt for baseline vs fine-tuned eval is reasonable
    system_prompt_base = "Answer the following medical question based on the provided context if available:"
    if context:
        prompt = f"[INST] {system_prompt_base}\n\nContext: {context}\n\nQuestion: {question} [/INST]"
    else:
        prompt = f"[INST] {system_prompt_base}\n\nQuestion: {question} [/INST]"


    original_padding_side = tokenizer_to_use.padding_side
    tokenizer_to_use.padding_side = "left"
    target_device = model_to_use.device if hasattr(model_to_use, 'device') else 'cuda:0'
```

```
    target_device = model_to_use.device if hasattr(model_to_use, "device") else "cuda:0"
    inputs = tokenizer_to_use(prompt, return_tensors="pt", truncation=True, max_length=1536).to(target_device)
    tokenizer_to_use.padding_side = original_padding_side # Reset padding side

    # Generation parameters
    with torch.no_grad():
        outputs = model_to_use.generate(
            **inputs,
            max_new_tokens=256, do_sample=True, temperature=0.7,
            top_p=0.9, repetition_penalty=1.1, no_repeat_ngram_size=3,
            pad_token_id=tokenizer_to_use.eos_token_id # Crucial
        )

    input_length = inputs.input_ids.shape[1]
    generated_ids = outputs[0, input_length:]
    response_text = tokenizer_to_use.decode(generated_ids, skip_special_tokens=True)

    response = extract_response_baseline(response_text)
    return response

print(f"\nRunning baseline evaluation on {BASELINE_EVAL_SAMPLES} test samples...")
baseline_predictions_text = []
baseline_references_text = []
baseline_predictions_decision = []
baseline_references_decision = []
baseline_evaluation_data = []

actual_baseline_eval_count = min(BASELINE_EVAL_SAMPLES, len(test_dataset_raw))

for i in tqdm(range(actual_baseline_eval_count)):
    example = test_dataset_raw[i] # Use RAW test examples
    question = example["question"]
    context_dict = example.get("context", {})
    contexts = context_dict.get("contexts", [])
    context = " ".join(contexts) if contexts else None
    reference_text = example["long_answer"] # Use original long_answer as reference text
    reference_decision = example["final_decision"] # Use original final_decision

    try:
        # Generate prediction using the BASE model
        prediction_text = generate_for_baseline(question, context, model_to_use=base_model, tokenizer_to_use=base_toke
        prediction_decision = extract_decision_baseline(prediction_text)

        baseline_predictions_text.append(prediction_text)
        baseline_references_text.append(reference_text) # Compare against raw long_answer
        baseline_predictions_decision.append(prediction_decision)
        baseline_references_decision.append(reference_decision)
        baseline_evaluation_data.append({"index": i, "prediction": prediction_text, "pred_decision": prediction_decisi

    except Exception as e:
        print(f"Error generating baseline prediction for index {i}: {e}")
        baseline_predictions_text.append("[ERROR]")
        baseline_references_text.append(reference_text)
        baseline_predictions_decision.append("error")
        baseline_references_decision.append(reference_decision)
        baseline_evaluation_data.append({"index": i, "prediction": "[ERROR]", "pred_decision": "error"})


# --- Calculate Baseline Metrics ---
print("\nCalculating baseline metrics...")
# Store results globally so they persist for the comparison cell later
global baseline_results # Declare as global
baseline_results = {}
if baseline_predictions_text:
    valid_indices = [i for i, p in enumerate(baseline_predictions_text) if p != "[ERROR]"]
    if valid_indices:
        valid_preds_text = [baseline_predictions_text[i] for i in valid_indices]
```

```
        valid_refs_text = [baseline_references_text[i] for i in valid_indices]
        valid_preds_decision = [baseline_predictions_decision[i] for i in valid_indices]
        valid_refs_decision = [baseline_references_decision[i] for i in valid_indices]

        print(f"Calculating metrics on {len(valid_indices)} valid baseline predictions...")

        try:
            rouge_scores = rouge_metric.compute(predictions=valid_preds_text, references=valid_refs_text, use_stemmer=
            baseline_results["baseline_rouge1"] = rouge_scores["rouge1"]
            baseline_results["baseline_rouge2"] = rouge_scores["rouge2"]
            baseline_results["baseline_rougeL"] = rouge_scores["rougeL"]
        except Exception as e: print(f"Error calculating baseline ROUGE: {e}")

        try:
            if len(valid_preds_text) > 0:
                 print("Calculating BERTScore (baseline)...")
                 bert_scores = bertscore_metric.compute(predictions=valid_preds_text, references=valid_refs_text, lang
                 baseline_results["baseline_bertscore_f1"] = np.mean(bert_scores["f1"]) if bert_scores.get("f1") else
                 print("BERTScore calculation finished.")
            else: baseline_results["baseline_bertscore_f1"] = 0.0
        except Exception as e: print(f"Error calculating baseline BERTScore: {e}")

        try:
            if len(valid_refs_decision) > 0 and len(valid_preds_decision) == len(valid_refs_decision):
                labels = sorted(list(set(valid_refs_decision + valid_preds_decision)))
                print(f"Decision labels found in baseline: {labels}")
                baseline_results["baseline_decision_f1_weighted"] = f1_score(valid_refs_decision, valid_preds_decision
                baseline_results["baseline_decision_f1_macro"] = f1_score(valid_refs_decision, valid_preds_decision, a
                baseline_results["baseline_decision_accuracy"] = accuracy_score(valid_refs_decision, valid_preds_decis
            else:
                 print("Not enough valid decision data for baseline F1/Accuracy calculation.")
                 baseline_results["baseline_decision_f1_weighted"]=0.0; baseline_results["baseline_decision_f1_macro"]
        except Exception as e: print(f"Error calculating baseline Decision Metrics: {e}")
    else:
        print("No valid baseline predictions generated.")
else:
    print("Baseline prediction list is empty.")

# --- Print Baseline Results ---
print("\n===== BASELINE EVALUATION RESULTS =====")
if baseline_results:
    for metric, score in baseline_results.items():
        if isinstance(score, (float, np.number, int)): print(f"{metric}: {score:.4f}")
        else: print(f"{metric}: {score}")
else:
    print("No baseline metrics calculated.")
```

```
--- Phase 1: Baseline Model Evaluation ---
Will evaluate baseline on 100 test samples.
Loading resources for baseline evaluation...
Downloading builder script: 100% 6.27k/6.27k [00:00<00:00, 669kB/s]
Downloading builder script: 100% 7.95k/7.95k [00:00<00:00, 928kB/s]
Loading base model: mistralai/Mistral-7B-v0.1
config.json: 100% 571/571 [00:00<00:00, 68.3kB/s]
model.safetensors.index.json: 100% 25.1k/25.1k [00:00<00:00, 2.26MB/s]
Fetching 2 files: 100% 2/2 [00:51<00:00, 51.20s/it]
```

```
model-00001-of-00002.safetensors: 100% 9.94G/9.94G [00:51<00:00, 217MB/s]
model-00002-of-00002.safetensors: 100% 4.54G/4.54G [00:27<00:00, 231MB/s]
Loading checkpoint shards: 100% 2/2 [00:17<00:00,  8.12s/it]
generation_config.json: 100% 116/116 [00:00<00:00, 14.2kB/s]
tokenizer_config.json: 100% 996/996 [00:00<00:00, 133kB/s]
tokenizer.model: 100% 493k/493k [00:00<00:00, 49.7MB/s]
tokenizer.json: 100% 1.80M/1.80M [00:00<00:00, 25.4MB/s]
special_tokens_map.json: 100% 414/414 [00:00<00:00, 56.0kB/s]
Base model and tokenizer loaded.

Running baseline evaluation on 100 test samples...
100% 100/100 [01:59<00:00,  1.14it/s]

Calculating baseline metrics...
Calculating metrics on 100 valid baseline predictions...
Calculating BERTScore (baseline)...
tokenizer_config.json: 100% 52.0/52.0 [00:00<00:00, 6.38kB/s]
config.json: 100% 792/792 [00:00<00:00, 101kB/s]
vocab.json: 100% 899k/899k [00:00<00:00, 25.6MB/s]
merges.txt: 100% 456k/456k [00:01<00:00, 344kB/s]
pytorch_model.bin: 100% 3.04G/3.04G [00:12<00:00, 244MB/s]
model.safetensors: 100% 3.04G/3.04G [00:15<00:00, 250MB/s]
BERTScore calculation finished.
Decision labels found in baseline: ['maybe', 'no', 'unknown', 'yes']

===== BASELINE EVALUATION RESULTS =====
baseline_rouge1: 0.0391
baseline_rouge2: 0.0043
baseline_rougeL: 0.0299
baseline_bertscore_f1: 0.1673
baseline_decision_f1_weighted: 0.2215
baseline_decision_f1_macro: 0.1289
```

Configuring and preparing the Mistral 7B model for **QLoRA** fine-tuning.

```
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
import torch
import gc


model_id = "mistralai/Mistral-7B-v0.1"

bnb_config = BitsAndBytesConfig(
```

```
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True
)

# --- Load Base Model with Quantization ---
print(f"Loading base model ({model_id}) with quantization for fine-tuning...")
# Clear potential leftover models from baseline eval
if 'base_model' in globals(): del base_model; gc.collect(); torch.cuda.empty_cache()
if 'ft_model' in globals(): del ft_model; gc.collect(); torch.cuda.empty_cache()
if 'model' in globals(): del model; gc.collect(); torch.cuda.empty_cache()


model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True
)

tokenizer = AutoTokenizer.from_pretrained(model_id)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"


model.gradient_checkpointing_enable()
model = prepare_model_for_kbit_training(model)

lora_r = 16                 # LoRA rank dimension
lora_alpha = 32             # LoRA scaling factor (often 2*r)
lora_dropout = 0.05         # Dropout probability for LoRA layers

peft_config = LoraConfig(
    r=lora_r,
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=[
        "q_proj",
        "k_proj",
        "v_proj",
        "o_proj",
        "gate_proj",
        "up_proj",
        "down_proj",
    ]
)

model = get_peft_model(model, peft_config)

def print_trainable_parameters(model):
    trainable_params = 0
    all_param = 0
    for _, param in model.named_parameters():
        all_param += param.numel()
        if param.requires_grad:
            trainable_params += param.numel()
    print(
        f"Trainable params: {trainable_params} || all params: {all_param} || trainable%: {100 * trainable_params / all
    )

print_trainable_parameters(model)
```

```
Loading base model (mistralai/Mistral-7B-v0.1) with quantization for fine-tuning...
```
Loading checkpoint shards: 100% 2/2 [00:17<00:00, 8.24s/it]
```
Trainable params: 41943040 || all params: 3794014208 || trainable%: 1.11
```

Tokenizing the text data.

```
from transformers import TrainingArguments, Trainer, DataCollatorForSeq2Seq
from datasets import load_from_disk


def tokenize_function_for_training(example):

    instruction_plus_input = example['instruction']
    response = example['output']

    text = f"<s>[INST] {instruction_plus_input} [/INST] {response}{tokenizer.eos_token}"

    # Tokenize the full text
    tokenized = tokenizer(
        text,
        truncation=True,
        padding=False,
        max_length=2048,
        return_tensors=None
    )


    # Tokenize only the instruction part
    prompt_part = f"<s>[INST] {instruction_plus_input} [/INST] "
    prompt_tokenized = tokenizer(prompt_part, truncation=True, max_length=2048, add_special_tokens=False) # Don't add
    prompt_length = len(prompt_tokenized['input_ids'])

    # Create labels: -100 for prompt tokens
    labels = [-100] * prompt_length + tokenized['input_ids'][prompt_length:]

    tokenized['labels'] = labels[:len(tokenized['input_ids'])]

    return tokenized

#Apply Tokenization to the Formatted Training Data
try:
    columns_to_remove = ["instruction", "input", "output"]
    tokenized_dataset = formatted_train_dataset.map(
        tokenize_function_for_training,
        batched=False, # Process one by one
        remove_columns=columns_to_remove
    )
    print("Tokenization applied successfully.")
except Exception as e:
    print(f"Error during tokenization: {e}")
    raise

tokenized_train_val_dataset = tokenized_dataset.train_test_split(test_size=0.1, seed=42) # Use 10% of training data fo

print(f"Tokenized Training set for Trainer: {len(tokenized_train_val_dataset['train'])}")
print(f"Tokenized Validation set for Trainer: {len(tokenized_train_val_dataset['test'])}")


data_collator = DataCollatorForSeq2Seq(
    tokenizer=tokenizer,
    padding=True, # Pad to longest sequence in batch
    return_tensors="pt", # Return PyTorch tensors
```

```
    pad_to_multiple_of=8 # Optional: Pad to multiple of 8 for better hardware utilization
)
print("Data collator configured.")
```


```
Tokenization applied successfully.
Tokenized Training set for Trainer: 810
Tokenized Validation set for Trainer: 90
Data collator configured.
```

Setup Training Arguments and Trainer

Defines the hyperparameters and configuration settings for the fine-tuning process.

Calculate metrics for camparison with the base model.

```
from transformers import TrainingArguments, Trainer
from transformers import Seq2SeqTrainingArguments, Trainer

print("Setting up Training Arguments and Trainer...")

# Ensure required variables are available
if 'model' not in globals(): raise NameError("Variable 'model' not found. Ensure QLoRA setup cell ran.")
if 'tokenizer' not in globals(): raise NameError("Variable 'tokenizer' not found.")
if 'tokenized_train_val_dataset' not in globals(): raise NameError("Variable 'tokenized_train_val_dataset' not found."
if 'data_collator' not in globals(): raise NameError("Variable 'data_collator' not found.")


print(dir(TrainingArguments))

output_dir = "./results_mistral_medical_v2"
training_args = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=4,
    learning_rate=1e-4,
    per_device_train_batch_size=2,
    per_device_eval_batch_size=2,
    gradient_accumulation_steps=8,
    weight_decay=0.01,
    optim="paged_adamw_8bit",
    warmup_ratio=0.03,
    lr_scheduler_type="cosine",
    fp16=True,
    bf16=False,
    eval_strategy="steps",
    eval_steps=50,
    save_strategy="steps",
    save_steps=100,
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
    logging_steps=10,
    gradient_checkpointing_kwargs={'use_reentrant': False},
    report_to="tensorboard",
    push_to_hub=False,
)
print("TrainingArguments defined.")
```

```
# Initialize the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train_val_dataset["train"],
    eval_dataset=tokenized_train_val_dataset["test"],
    data_collator=data_collator,
    tokenizer=tokenizer)
print("Trainer initialized.")

# Start training
print("\nStarting fine-tuning...")
try:
    train_result = trainer.train()
    print("Fine-tuning finished.")

    # Save training metrics
    metrics = train_result.metrics
    trainer.log_metrics("train", metrics)
    trainer.save_metrics("train", metrics)

    # Save the final model adapters
    final_model_path = "./final_model_mistral_medical" # Define path for final adapters
    print(f"\nSaving final model adapters to {final_model_path}...")
    trainer.save_model(final_model_path) # Saves the LoRA adapters
    # tokenizer.save_pretrained(final_model_path) # Optionally save tokenizer config with adapters
    print("Final model adapters saved.")



except Exception as e:
    print(f"An error occurred during training: {e}")
    raise

print("\n--- End of Phase 2 ---")
```

```
Setting up Training Arguments and Trainer...
['_VALID_DICT_FIELDS', '__annotations__', '__class__', '__dataclass_fields__', '__datacla
TrainingArguments defined.
<ipython-input-9-8708fa622083>:44: FutureWarning: `tokenizer` is deprecated and will be r
  trainer = Trainer(
No label_names provided for model class `PeftModelForCausalLM`. Since `PeftModel` hides b
Trainer initialized.

Starting fine-tuning...
`use_cache=True` is incompatible with gradient checkpointing. Setting `use_cache=False`.
/usr/local/lib/python3.11/dist-packages/torch/_dynamo/eval_frame.py:745: UserWarning: tor
  return fn(*args, **kwargs)
```

[200/200 22:38, Epoch 3/4]

Step	Training Loss	Validation Loss
50	1.247800	1.131244
100	0.940000	1.206602
150	0.552200	1.403527
200	0.316100	1.646750

```
/usr/local/lib/python3.11/dist-packages/torch/_dynamo/eval_frame.py:745: UserWarning: tor
  return fn(*args, **kwargs)
Fine-tuning finished.
***** train metrics *****
```

```
    epoch                    =     3.9877
    total_flos               = 75442215GF
    train_loss               =     0.8138
    train_runtime            = 0:22:45.50
    train_samples_per_second =      2.373
    train_steps_per_second   =      0.146

  Saving final model adapters to ./final_model_mistral_medical...
  Final model adapters saved.

  --- End of Phase 2 ---
```

Fine-Tuned Model Evaluation & Comparison

```
import numpy as np
import pandas as pd
import evaluate
from tqdm.auto import tqdm
import re
import torch
from sklearn.metrics import f1_score, accuracy_score
import warnings
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
from peft import PeftModel
import gc

warnings.filterwarnings("ignore", category=UserWarning, module='sklearn') # Suppress potential warnings

FINETUNED_EVAL_SAMPLES = len(test_dataset_raw)

FINETUNED_MODEL_PATH = "./final_model_mistral_medical"

OUTPUT_FILENAME_COMPARISON = "comparison_evaluation_results.csv"

try:
    rouge_metric = evaluate.load("rouge")
    bertscore_metric = evaluate.load("bertscore")

    base_model_id_ft = "mistralai/Mistral-7B-v0.1"
    bnb_config_ft = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.float16,
        bnb_4bit_use_double_quant=True
    )

    # Now load the base model for the fine-tuned version
    print(f"Loading base model ({base_model_id_ft}) directly onto GPU...")
    base_model_ft = AutoModelForCausalLM.from_pretrained(
        base_model_id_ft,
        quantization_config=bnb_config_ft,
        device_map="cuda:0", # Force loading onto the first GPU
        trust_remote_code=True
    )
    # Load the tokenizer associated with the base model
    ft_tokenizer = AutoTokenizer.from_pretrained(base_model_id_ft)
    if ft_tokenizer.pad_token is None:
        ft_tokenizer.pad_token = ft_tokenizer.eos_token
    ft_tokenizer.padding_side = "left" # Use left padding for generation

    print(f"Loading fine-tuned LoRA adapters from: {FINETUNED_MODEL_PATH}")
```

```
    # Load the PEFT model by combining base model and adapters
    ft_model = PeftModel.from_pretrained(base_model_ft, FINETUNED_MODEL_PATH)

    # Merge the adapters into the base model
    print("Merging LoRA adapters...")
    ft_model = ft_model.merge_and_unload()
    print("Fine-tuned model loaded and merged.")


except Exception as e:
    print(f"Error loading resources for fine-tuned eval: {e}")
    raise # Reraise the exception after printing context

if 'SYSTEM_PROMPT' not in globals():
     SYSTEM_PROMPT = """You are a medical assistant trained to answer health related questions based on medical Data.
Use the provided context from the dataset to give accurate, helpful responses.
Base your answers only on the information provided in the context, and if you're unsure, acknowledge the limitations o
     print("Redefined SYSTEM_PROMPT.")

if 'format_instruction' not in globals():
    def format_instruction(example):
         q = example.get("question", "")
         cd = example.get("context", {})
         cs = cd.get("contexts", [])
         fc = " ".join(cs) if cs else ""
         a = example.get("long_answer", "")
         fd = example.get("final_decision", "unknown")
         instr = f"{SYSTEM_PROMPT}\n\nQuestion: {q}\n\nContext: {fc}"
         resp = f"{a} The direct answer to your question is: {fd}."
         return {"instruction": instr, "input": "", "output": resp}
    print("Redefined format_instruction function for reference generation.")

def extract_response_ft(text):
    text = text.strip()
    # Find the *last* occurrence of [/INST]
    inst_end_pos = text.rfind("[/INST]")
    if inst_end_pos != -1: response = text[inst_end_pos + len("[/INST]"):].strip()
    else: response = text
    response = response.replace("<s>", "").replace("</s>", "").strip()
    return response

def extract_decision_ft(text):
    if not text or not text.strip(): return "unknown"
    text_lower = text.lower()
    # Look for the explicit format first
    match = re.search(r"the direct answer.*? is:\s*(yes|no|maybe)", text_lower)
    if match: return match.group(1)
    # Fallback check at the beginning
    first_part = text_lower[:150]
    if "yes" in first_part: return "yes"
    if "no" in first_part: return "no"
    if "maybe" in first_part: return "maybe"
    return "unknown"

def generate_for_finetuned(question, context=None, model_to_use=None, tokenizer_to_use=None):
    if model_to_use is None or tokenizer_to_use is None: raise ValueError("Model/Tokenizer needed.")
    # Use the same SYSTEM_PROMPT that was used to format the training data
    SYSTEM_PROMPT_FT = SYSTEM_PROMPT

    if context: prompt = f"<s>[INST] {SYSTEM_PROMPT_FT}\n\nQuestion: {question}\n\nContext: {context} [/INST]"
    else: prompt = f"<s>[INST] {SYSTEM_PROMPT_FT}\n\nQuestion: {question} [/INST]"

    original_padding_side = tokenizer_to_use.padding_side
    tokenizer_to_use.padding_side = "left"
    target_device = model_to_use.device if hasattr(model_to_use, 'device') else 'cuda:0'
    inputs = tokenizer_to_use(prompt, return_tensors="pt", truncation=True, max_length=1536).to(target_device)
```

```
    tokenizer_to_use.padding_side = original_padding_side

    with torch.no_grad():
        outputs = model_to_use.generate(
            **inputs, max_new_tokens=256, do_sample=True, temperature=0.6,
            top_p=0.9, repetition_penalty=1.15, no_repeat_ngram_size=3,
            pad_token_id=tokenizer_to_use.eos_token_id
        )
    input_length = inputs.input_ids.shape[1]
    generated_ids = outputs[0, input_length:]
    response_text = tokenizer_to_use.decode(generated_ids, skip_special_tokens=True)
    response = extract_response_ft(response_text)
    return response

print(f"\nRunning fine-tuned evaluation on {FINETUNED_EVAL_SAMPLES} test samples...")
ft_predictions_text = []
ft_references_text = []
ft_predictions_decision = []
ft_references_decision = []
ft_evaluation_data = []

actual_ft_eval_count = min(FINETUNED_EVAL_SAMPLES, len(test_dataset_raw))

for i in tqdm(range(actual_ft_eval_count)):
    example = test_dataset_raw[i] # Use RAW test data
    question = example["question"]
    context_dict = example.get("context", {})
    contexts = context_dict.get("contexts", [])
    context = " ".join(contexts) if contexts else None

    reference_obj = format_instruction(example)
    reference_text_ft = reference_obj["output"]
    reference_decision_ft = extract_decision_ft(reference_text_ft)

    try:
        # Generate using the FINE-TUNED model (ft_model)
        prediction_text_ft = generate_for_finetuned(question, context, model_to_use=ft_model, tokenizer_to_use=ft_toke
        prediction_decision_ft = extract_decision_ft(prediction_text_ft) # Use stricter extraction

        ft_predictions_text.append(prediction_text_ft)
        ft_references_text.append(reference_text_ft) # Compare against formatted reference
        ft_predictions_decision.append(prediction_decision_ft)
        ft_references_decision.append(reference_decision_ft)

        # Store detailed results
        ft_evaluation_data.append({
            "index": i, "question": question,
            "reference_text": reference_text_ft, "prediction_text": prediction_text_ft,
            "reference_decision": reference_decision_ft, "prediction_decision": prediction_decision_ft,
            "decision_correct": prediction_decision_ft == reference_decision_ft
        })

    except Exception as e:
        print(f"Error generating fine-tuned prediction for index {i}: {e}")
        ft_predictions_text.append("[ERROR]")
        ft_references_text.append(reference_text_ft)
        ft_predictions_decision.append("error")
        ft_references_decision.append(reference_decision_ft)
        ft_evaluation_data.append({ "index": i, "question": "Error", "reference_text": reference_text_ft, "prediction_

print("\nCalculating fine-tuned metrics...")
finetuned_results = {}
if ft_predictions_text:
    valid_indices_ft = [i for i, p in enumerate(ft_predictions_text) if p != "[ERROR]"]
    if valid_indices_ft:
        valid_preds_text_ft = [ft_predictions_text[i] for i in valid_indices_ft]
```

```
        valid_refs_text_ft = [ft_references_text[i] for i in valid_indices_ft]
        valid_preds_decision_ft = [ft_predictions_decision[i] for i in valid_indices_ft]
        valid_refs_decision_ft = [ft_references_decision[i] for i in valid_indices_ft]
        print(f"Calculating metrics on {len(valid_indices_ft)} valid fine-tuned predictions...")
        try:
            rouge_scores_ft = rouge_metric.compute(predictions=valid_preds_text_ft, references=valid_refs_text_ft, use
            finetuned_results["finetuned_rouge1"] = rouge_scores_ft["rouge1"]
            finetuned_results["finetuned_rouge2"] = rouge_scores_ft["rouge2"]
            finetuned_results["finetuned_rougeL"] = rouge_scores_ft["rougeL"]
        except Exception as e: print(f"Error calculating fine-tuned ROUGE: {e}")
        try:
             if len(valid_preds_text_ft) > 0:
                  print("Calculating BERTScore (this may take a while)...")
                  target_device_bert = ft_model.device if hasattr(ft_model, 'device') and ft_model.device.type == 'cud
                  bert_scores_ft = bertscore_metric.compute(predictions=valid_preds_text_ft, references=valid_refs_tex
                  finetuned_results["finetuned_bertscore_f1"] = np.mean(bert_scores_ft["f1"]) if bert_scores_ft.get("f
                  print("BERTScore calculation finished.")
             else: finetuned_results["finetuned_bertscore_f1"] = 0.0
        except Exception as e: print(f"Error calculating fine-tuned BERTScore: {e}")
        try:
             if len(valid_refs_decision_ft) > 0 and len(valid_preds_decision_ft) == len(valid_refs_decision_ft):
                  labels_ft = sorted(list(set(valid_refs_decision_ft + valid_preds_decision_ft)))
                  print(f"Decision labels found in fine-tuned: {labels_ft}")
                  finetuned_results["finetuned_decision_f1_weighted"] = f1_score(valid_refs_decision_ft, valid_preds_d
                  finetuned_results["finetuned_decision_f1_macro"] = f1_score(valid_refs_decision_ft, valid_preds_deci
                  finetuned_results["finetuned_decision_accuracy"] = accuracy_score(valid_refs_decision_ft, valid_pred
             else:
                  print("Not enough valid decision data for fine-tuned F1/Accuracy calculation.")
                  finetuned_results["finetuned_decision_f1_weighted"] = 0.0; finetuned_results["finetuned_decision_f1_
        except Exception as e: print(f"Error calculating fine-tuned Decision Metrics: {e}")
    else: print("No valid fine-tuned predictions generated.")
else: print("Fine-tuned prediction list is empty.")

if 'baseline_results' not in globals():
    print("Warning: 'baseline_results' ")
    baseline_results = {}

if not baseline_results: # If baseline results are missing or empty
     comparison_df = pd.DataFrame([{"Metric": k.replace('finetuned_','').title(), "Fine-tuned": f"{v:.4f}" if isinstan
     print("Displaying only Fine-tuned results as Baseline results are unavailable.")
else: # Proceed with comparison if baseline_results exist
    comparison_data = []
    metric_keys_base = {k.replace('baseline_', '') for k in baseline_results.keys()}
    metric_keys_ft = {k.replace('finetuned_', '') for k in finetuned_results.keys()}
    all_metrics = sorted(list(metric_keys_base.union(metric_keys_ft)))
    for metric in all_metrics:
        baseline_key = f"baseline_{metric}"
        finetuned_key = f"finetuned_{metric}"
        base_score = baseline_results.get(baseline_key)
        ft_score = finetuned_results.get(finetuned_key)
        improvement = None
        is_base_num = isinstance(base_score, (float, np.number, int))
        is_ft_num = isinstance(ft_score, (float, np.number, int))
        if is_base_num and is_ft_num:
            improvement = ft_score - base_score
            base_str = f"{base_score:.4f}"; ft_str = f"{ft_score:.4f}"; imp_str = f"{improvement:+.4f}"
        else: # Handle cases where scores might not be numbers
            base_str = f"{base_score:.4f}" if is_base_num else str(base_score); ft_str = f"{ft_score:.4f}" if is_ft_nu
        comparison_data.append({ "Metric": metric.replace('_', ' ').title(), "Baseline": base_str, "Fine-tuned": ft_st
    comparison_df = pd.DataFrame(comparison_data)

# Display the comparison table
print(comparison_df.to_string(index=False))


if ft_evaluation_data:
```

```
    try:
        eval_df_ft = pd.DataFrame(ft_evaluation_data)
        eval_df_ft.to_csv(OUTPUT_FILENAME_COMPARISON, index=False)
        print(f"\nDetailed fine-tuned evaluation results saved to {OUTPUT_FILENAME_COMPARISON}")
    except Exception as e:
        print(f"Error saving detailed results: {e}")
```

```
Loading base model (mistralai/Mistral-7B-v0.1) directly onto GPU...
Loading checkpoint shards: 100% 2/2 [00:17<00:00,  7.99s/it]
Loading fine-tuned LoRA adapters from: ./final_model_mistral_medical
Merging LoRA adapters...
/usr/local/lib/python3.11/dist-packages/peft/tuners/lora/bnb.py:351: UserWarning: Merge l
  warnings.warn(
Fine-tuned model loaded and merged.

Running fine-tuned evaluation on 100 test samples...
100% 100/100 [03:57<00:00,  2.74s/it]

Calculating fine-tuned metrics...
Calculating metrics on 100 valid fine-tuned predictions...
Calculating BERTScore (this may take a while)...
BERTScore calculation finished.
Decision labels found in fine-tuned: ['maybe', 'no', 'unknown', 'yes']
              Metric Baseline Fine-tuned Improvement
        Bertscore F1   0.1673     0.5360     +0.3687
   Decision Accuracy   0.1500     0.2500     +0.1000
   Decision F1 Macro   0.1289     0.1548     +0.0259
Decision F1 Weighted   0.2215     0.2540     +0.0325
              Rouge1   0.0391     0.1940     +0.1549
              Rouge2   0.0043     0.0317     +0.0274
              Rougel   0.0299     0.1314     +0.1015

Detailed fine-tuned evaluation results saved to comparison_evaluation_results.csv
Warning: Empty candidate sentence detected; setting raw BERTscores to 0.
Warning: Empty candidate sentence detected; setting raw BERTscores to 0.
```

```
import torch
from IPython.display import display, HTML, clear_output
import ipywidgets as widgets
import gc
import re

# --- Create Demo Interface ---
def create_comparison_demo_interface():
    title = widgets.HTML(value="<h2>Medical AI Assistant Demo (Comparison)</h2><p>Enter a medical question to compare
    question_input = widgets.Textarea(
        value='', placeholder='Enter a medical question...', description='Question:', disabled=False,
        layout=widgets.Layout(width='95%', height='80px')
    )
    context_input = widgets.Textarea(
        value='', placeholder='(Optional) Add context from a medical paper...', description='Context:', disabled=False
        layout=widgets.Layout(width='95%', height='100px')
    )
    submit_button = widgets.Button(description='Compare Responses', button_style='success', tooltip='Click to generate
    output_area = widgets.Output(layout={'border': '1px solid #ddd', 'padding': '10px', 'margin_top': '10px'})

    # --- Button Click Handler ---
    def on_button_clicked(b):
```

```
        with output_area:
            clear_output()
            question = question_input.value.strip()
            context = context_input.value.strip() or None

            if not question:
                print("Please enter a question.")
                return

            print("Generating response from Fine-tuned model...")
            try:
                # Use the generation function designed for the fine-tuned model
                ft_response = generate_for_finetuned(question, context, model_to_use=ft_model, tokenizer_to_use=ft_tok

                ft_response_html = ft_response.replace('<', '&lt;').replace('>', '&gt;').replace('\n', '<br>')
            except Exception as e:
                print(f"Error generating fine-tuned response: {e}")
                ft_response_html = f"<i style='color:red;'>Error: {e}</i>"

            print("Generating response from Base model...")
            try:
                # Use the generation function designed for the baseline model
                base_response = generate_for_baseline(question, context, model_to_use=base_model, tokenizer_to_use=bas
                base_response_html = base_response.replace('<', '&lt;').replace('>', '&gt;').replace('\n', '<br>')
            except Exception as e:
                print(f"Error generating baseline response: {e}")
                base_response_html = f"<i style='color:red;'>Error: {e}</i>"

            print("Displaying results...")
            display(HTML(f"""
            <div style="display: flex; flex-direction: row; width: 100%; border-top: 1px solid #eee; padding-top: 10px
                <div style="flex: 1; padding-right: 10px; border-right: 1px solid #eee;">
                    <h4>Fine-tuned Model Response:</h4>

                    <div style="background-color: #e7f5fe; padding: 10px; border-radius: 5px; min-height: 100px; overf
                        {ft_response_html}
                    </div>
                </div>
                <div style="flex: 1; padding-left: 10px;">
                    <h4>Base Model Response:</h4>

                    <div style="background-color: #f8f9fa; padding: 10px; border-radius: 5px; min-height: 100px; overf
                        {base_response_html}
                    </div>
                </div>
            </div>
            """))

    submit_button.on_click(on_button_clicked)

    # Assemble the UI

    return widgets.VBox([title, question_input, submit_button, output_area])

comparison_demo = create_comparison_demo_interface()
display(comparison_demo)
```

Medical AI Assistant Demo (Comparison)

Enter a medical question to compare Fine-tuned vs. Base Mistral 7B responses.

Question: Enter a medical question...

Compare Respons...

```
!pip install gradio
import gradio as gr
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
import gc
import os

# Suppress unnecessary output
import warnings
warnings.filterwarnings("ignore")

# Clear memory
if 'model' in globals():
    del model
    gc.collect()
    torch.cuda.empty_cache()

# Load model function - no print statements
def load_medical_model(model_path="./final_model_mistral_medical"):
    # Check if path exists
    if not os.path.exists(model_path):
        raise FileNotFoundError(f"Model path {model_path} not found.")

    # Load tokenizer
    tokenizer = AutoTokenizer.from_pretrained(model_path, local_files_only=True)
    if tokenizer.pad_token is None:
        tokenizer.pad_token = tokenizer.eos_token

    # Load model on CPU
    model = AutoModelForCausalLM.from_pretrained(
        model_path,
        device_map="cpu",
        torch_dtype=torch.float32,
        local_files_only=True,
        low_cpu_mem_usage=True
    )
    model.eval()
    return model, tokenizer

# Fixed generate response function that returns proper format for chatbot
def generate_conversation_response(message, history):
    try:
        # Format message with conversation history
        system_prompt = "You are a medical assistant trained to answer health related questions based on medical data. Y

        # Create a conversation context from history
        conversation_context = ""
        if history:
            for human, ai in history[-3:]:  # Use last 3 exchanges for context
                conversation_context += f"Human: {human}\nAI: {ai}\n"

        # Build prompt with history
        prompt = f"<s>[INST] {system_prompt}\n\nConversation history:\n{conversation_context}\nHuman: {message}\nAI: [/I

        # Generate response
        inputs = tokenizer(prompt, return_tensors="pt", truncation=True, max_length=768)
        inputs = {k: v.to('cpu') for k, v in inputs.items()}
```

```
        with torch.no_grad():
            outputs = model.generate(
                **inputs,
                max_new_tokens=256,
                do_sample=True,
                temperature=0.7,
                top_p=0.9,
                repetition_penalty=1.1,
                pad_token_id=tokenizer.eos_token_id
            )

        # Extract response
        full_output = tokenizer.decode(outputs[0], skip_special_tokens=True)

        # Find response after instruction
        if "[/INST]" in full_output:
            response = full_output.split("[/INST]")[-1].strip()
        else:
            response = full_output

        # This is the key change - return updated history with new message pair
        history = history + [(message, response)]
        return history

    except Exception as e:
        # Also maintain correct format in case of error
        error_msg = f"I'm having trouble processing that request. Please try again or ask another question."
        history = history + [(message, error_msg)]
        return history

# Create chatbot interface
def create_medical_chatbot():
    with gr.Blocks(css="footer {visibility: hidden}") as demo:
        gr.Markdown("# Medical Conversation Assistant")
        gr.Markdown("Chat with an AI medical assistant trained to answer health-related questions.")

        chatbot = gr.Chatbot(
            height=500,
            bubble_full_width=False,
            avatar_images=(None, ""),
            show_label=False,
            elem_id="medical_chat"
        )

        msg = gr.Textbox(
            placeholder="Ask a medical question...",
            show_label=False,
            container=False
        )

        with gr.Row():
            submit_btn = gr.Button("Send", variant="primary")
            clear_btn = gr.Button("Clear Chat")

        # Set up events
        submit_btn.click(
            fn=generate_conversation_response,
            inputs=[msg, chatbot],
            outputs=chatbot,
            queue=True
        ).then(
            fn=lambda: "",
            inputs=None,
            outputs=msg
        )
```

```
        msg.submit(
            fn=generate_conversation_response,
            inputs=[msg, chatbot],
            outputs=chatbot,
            queue=True
        ).then(
            fn=lambda: "",
            inputs=None,
            outputs=msg
        )

        clear_btn.click(
            fn=lambda: [],  # Return empty list to clear chat
            inputs=None,
            outputs=chatbot,
            queue=False
        )

    return demo

# Main execution - minimal output
try:
    # Load model silently
    model, tokenizer = load_medical_model()

    # Launch the interface
    demo = create_medical_chatbot()
    demo.launch(share=True, debug=False)

except Exception as e:
    print(f"Error: {str(e)}")
```

```
Requirement already satisfied: gradio in /usr/local/lib/python3.11/dist-packages (5.29.0)
Requirement already satisfied: aiofiles<25.0,>=22.0 in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: anyio<5.0,>=3.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: fastapi<1.0,>=0.115.2 in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: ffmpy in /usr/local/lib/python3.11/dist-packages (from gra
Requirement already satisfied: gradio-client==1.10.0 in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: groovy~=0.1 in /usr/local/lib/python3.11/dist-packages (fr
Requirement already satisfied: httpx>=0.24.1 in /usr/local/lib/python3.11/dist-packages (
Requirement already satisfied: huggingface-hub>=0.28.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: jinja2<4.0 in /usr/local/lib/python3.11/dist-packages (fro
Requirement already satisfied: markupsafe<4.0,>=2.0 in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: numpy<3.0,>=1.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: orjson~=3.0 in /usr/local/lib/python3.11/dist-packages (fr
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: pandas<3.0,>=1.0 in /usr/local/lib/python3.11/dist-package
Requirement already satisfied: pillow<12.0,>=8.0 in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: pydantic<2.12,>=2.0 in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: pydub in /usr/local/lib/python3.11/dist-packages (from gra
Requirement already satisfied: python-multipart>=0.0.18 in /usr/local/lib/python3.11/dist
Requirement already satisfied: pyyaml<7.0,>=5.0 in /usr/local/lib/python3.11/dist-package
Requirement already satisfied: ruff>=0.9.3 in /usr/local/lib/python3.11/dist-packages (fr
Requirement already satisfied: safehttpx<0.2.0,>=0.1.6 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: semantic-version~=2.0 in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: starlette<1.0,>=0.40.0 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: tomlkit<0.14.0,>=0.12.0 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: typer<1.0,>=0.12 in /usr/local/lib/python3.11/dist-package
Requirement already satisfied: typing-extensions~=4.0 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: uvicorn>=0.14.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (from gr
Requirement already satisfied: websockets<16.0,>=10.0 in /usr/local/lib/python3.11/dist-p
```

```
Requirement already satisfied: idna>=2.8 in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: certifi in /usr/local/lib/python3.11/dist-packages (from h
Requirement already satisfied: httpcore==1.* in /usr/local/lib/python3.11/dist-packages (
Requirement already satisfied: h11>=0.16 in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: pydantic-core==2.33.2 in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: typing-inspection>=0.4.0 in /usr/local/lib/python3.11/dist
Requirement already satisfied: click>=8.0.0 in /usr/local/lib/python3.11/dist-packages (f
Requirement already satisfied: shellingham>=1.3.0 in /usr/local/lib/python3.11/dist-packa
Requirement already satisfied: rich>=10.11.0 in /usr/local/lib/python3.11/dist-packages (
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packa
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.11/dist-packages (fro
```

Loading checkpoint shards: 100% 2/2 [00:04<00:00, 2.04s/it]

```
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://505bdc0503750fed17.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `grad
```

No interface is running right now

Start coding or generate with AI.