Fine-Tuning Large Language Models for Enterprise Applications

---

1. Student Information

- **Student Name: Christo Raju**
- **Student ID (optional): S223880826**
- **Date Submitted: 16/05/2025**

---

2. Project Introduction

- **Title of the Project: Fine-Tuning LLMs for Enterprise**
- **What is the project about?**

  This project is about customizing large language models (LLMs) to work better for specific business tasks such as answering customer questions, generating content, or analyzing data. We will learn how to fine-tune these models using methods like Supervised Fine-Tuning (SFT) and Low-Rank Adaptation (LoRA), which help improve performance without needing huge computing power. The project involves creating custom datasets, using tools like Hugging Face Transformers and testing how well the updated models perform using standard metrics like BLEU, ROUGE, F1 and BERTScore. The goal is to achieve hands-on experience in adapting AI models for real-world enterprise use cases and documenting the process and results as best practices.

- **Why is this project important or useful?**

  Large Language Models (LLMs) are powerful, but they often need fine-tuning to perform well in specialized areas like healthcare, finance, or customer support. By learning how to fine-tune these models efficiently, we can make them more accurate, faster, and cost-effective for real-world applications. This not only improves the quality of enterprise solutions but also gives us valuable skills that are in high demand in the AI industry.

---

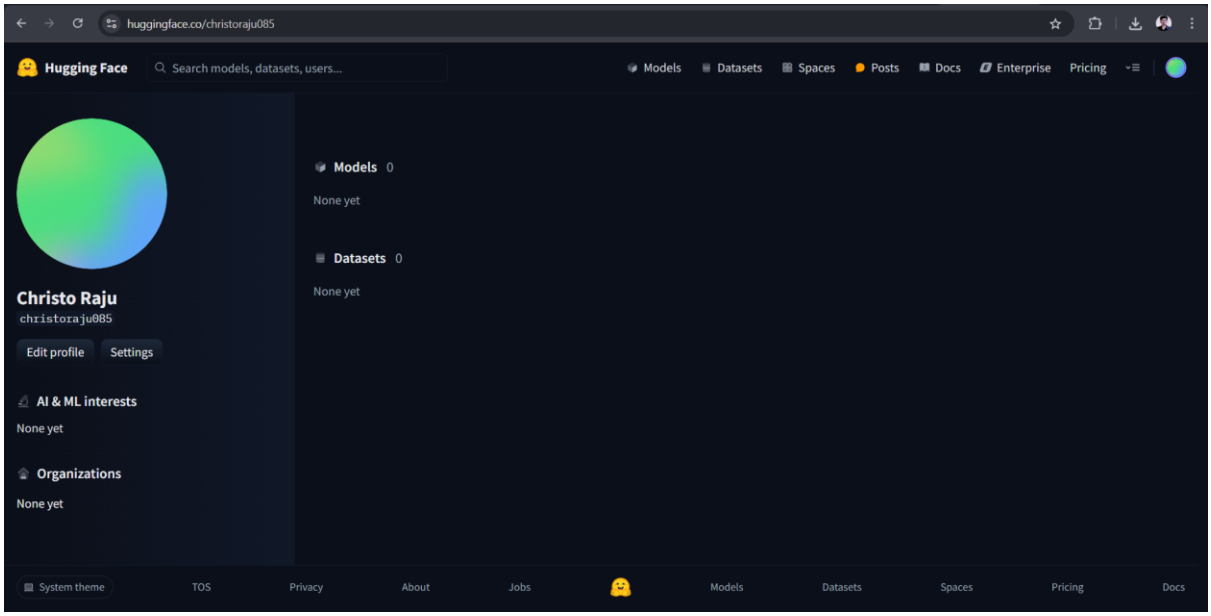
3. API/Token Setup — Hugging Face

Objective:

The goal of this step is to safely connect the Python code to my Hugging Face account so that i can download models and datasets. To do this, we create a secret access token on the Hugging Face website and use it in our code to log in. This helps make sure only we can access the Hugging Face resources securely.

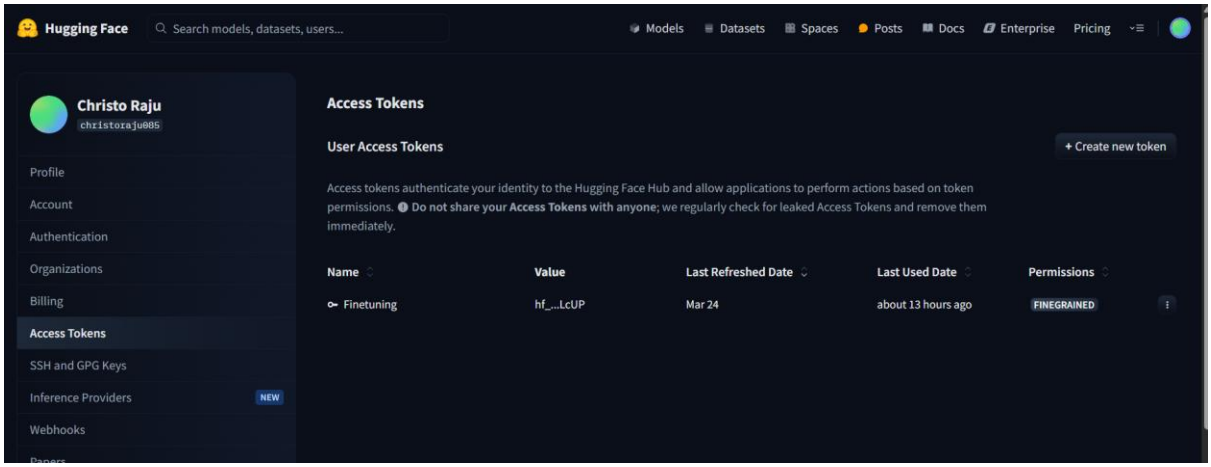
Step 1:

Created account on Hugging Face - https://huggingface.co/christoraju085



Step 2:

I logged in to my Hugging Face account. Then, I went to the Settings page and clicked on the “Access Tokens” section.

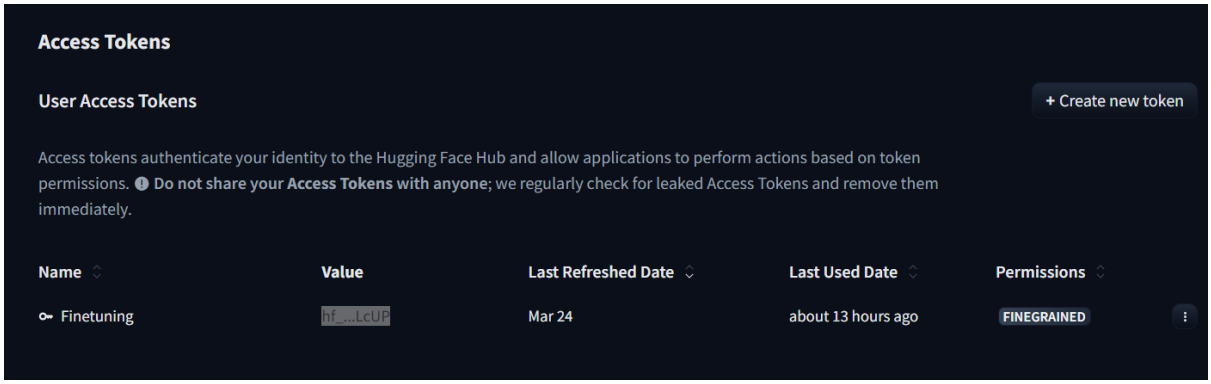


Step 3:

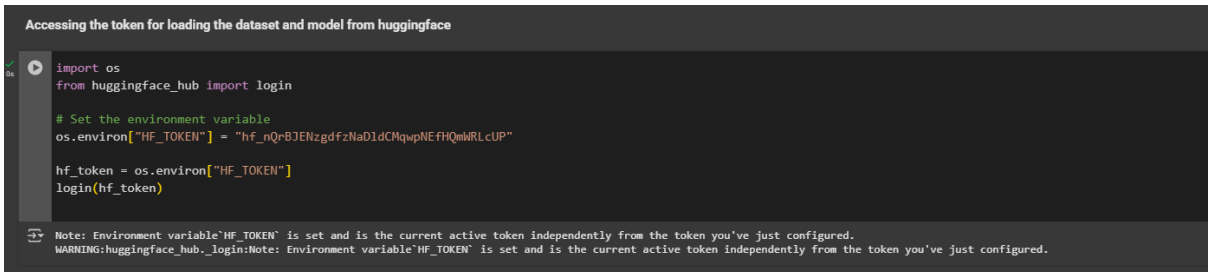
To create a new token, I clicked the “+ Create new token” button. I chose the right permissions for my project (like “Read” or “Fine-Grained”), gave the token a name as “Finetuning”

Step 4 :

Then I copied the token value and saved it in a safe place. This token helps the code connect to my Hugging Face account securely.



Screenshot or terminal output (required):



In this step, we securely logged in to Hugging Face using an environment variable instead of directly writing our token in the code. We used Python's os.environ to temporarily store the token as "HF_TOKEN", and then used that token to log in using the login() function from the huggingface_hub library.

The message shown is just a warning, saying that the environment variable HF_TOKEN is already active and being used, even before calling login() manually. It's not an error — it simply means Hugging Face has already picked up the token from the environment and is using it.

---

4. Environment Setup

Google Colab
Windows
GPU is not available

- **Python Version: Python 3.11.12**

Code: Environment & GPU Check

This code is used to check whether my computer has a GPU (graphics processing unit) available for running machine learning tasks. It uses PyTorch to look for a CUDA-compatible GPU. If a GPU is found, the code sets the device to use the GPU; otherwise, it defaults to using the CPU (central processing unit). Finally, it prints out which device is being used. In this case, the output shows "cpu", which means there is no GPU available, so the system will use the CPU to run the code.

---

5. LLM Setup

- **Model Name (e.g., GPT-3.5, LLaMA 2, Falcon, etc.): FLAN-T5 XL**
- **Provider (OpenAI, Hugging Face, etc.): Hugging Face**
- **Key Libraries & Dependencies (with versions):**

Library	**Version (if known)**	**Purpose**
transformers	4.51.3	Pretrained LLMs, tokenization, training APIs
datasets	2.14.4	Loading, preprocessing, and handling datasets
peft	0.15.2	Parameter-Efficient Fine-Tuning (used for LoRA)
bitsandbytes	0.45.5	Enables 8-bit/4-bit quantization (optional for model size optimization)
evaluate	0.4.3	Model evaluation metrics like BLEU, ROUGE, BERTScore
rouge_score	0.1.2	Used for ROUGE metric calculations
bert_score	0.3.12	Used for semantic similarity evaluation
gradio	5.29.0	Creating the interactive chatbot UI
pandas	2.2.2	Data manipulation and preprocessing

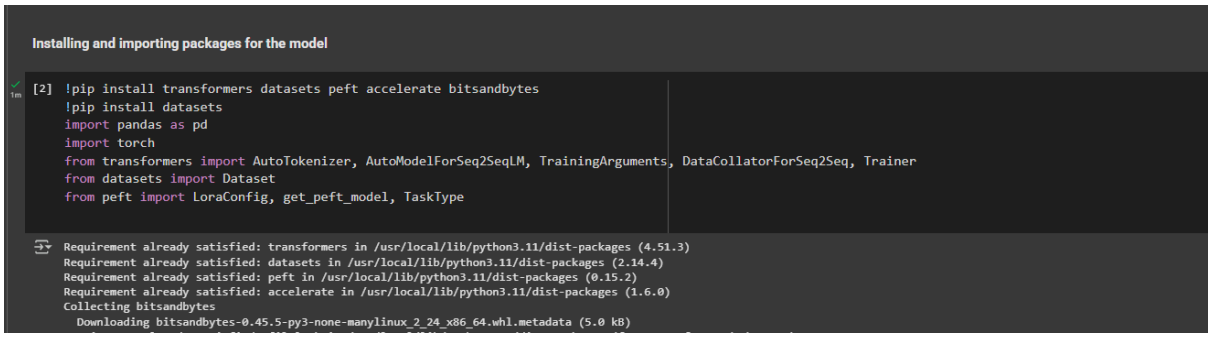
numpy	2.0.2	General numerical operations
matplotlib	3.10.0	Data visualization
torch	2.6.0	Deep learning framework
scikit-learn	1.6.1	Train/test splitting of benchmark dataset

```python
import transformers
import datasets
import peft
import bitsandbytes as bnb
import evaluate
import rouge_score
import bert_score
import gradio
import pandas as pd
import numpy as np
import matplotlib
import torch
import sklearn

print("transformers:", transformers.__version__)
print("datasets:", datasets.__version__)
print("peft:", peft.__version__)
print("bitsandbytes:", bnb.__version__)
print("evaluate:", evaluate.__version__)
print("bert_score:", bert_score.__version__)
print("gradio:", gradio.__version__)
print("pandas:", pd.__version__)
print("numpy:", np.__version__)
print("matplotlib:", matplotlib.__version__)
print("torch:", torch.__version__)
print("scikit-learn:", sklearn.__version__)
```

```
transformers: 4.51.3
datasets: 2.14.4
peft: 0.15.2
bitsandbytes: 0.45.5
evaluate: 0.4.3
bert_score: 0.3.12
gradio: 5.29.0
pandas: 2.2.2
numpy: 2.0.2
matplotlib: 3.10.0
torch: 2.6.0+cu124
scikit-learn: 1.6.1
```

Code: Install & Import

Transformer, Datasets, Peft, Pandas, Torch

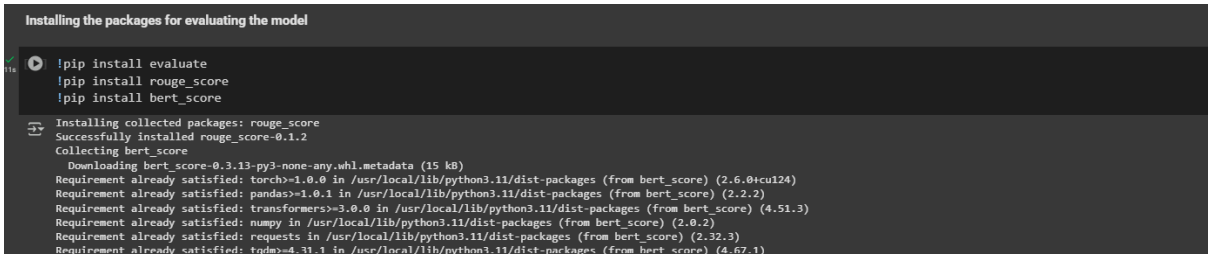


Installing and importing packages for the model

```
[2] !pip install transformers datasets peft accelerate bitsandbytes
    !pip install datasets
    import pandas as pd
    import torch
    from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, TrainingArguments, DataCollatorForSeq2Seq, Trainer
    from datasets import Dataset
    from peft import LoraConfig, get_peft_model, TaskType
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-packages (4.51.3)
Requirement already satisfied: datasets in /usr/local/lib/python3.11/dist-packages (2.14.4)
Requirement already satisfied: peft in /usr/local/lib/python3.11/dist-packages (0.15.2)
Requirement already satisfied: accelerate in /usr/local/lib/python3.11/dist-packages (1.6.0)
Collecting bitsandbytes
  Downloading bitsandbytes-0.45.5-py3-none-manylinux_2_24_x86_64.whl.metadata (5.0 kB)
```

Evaluate, Rouge Score, Bert Score



Installing the packages for evaluating the model

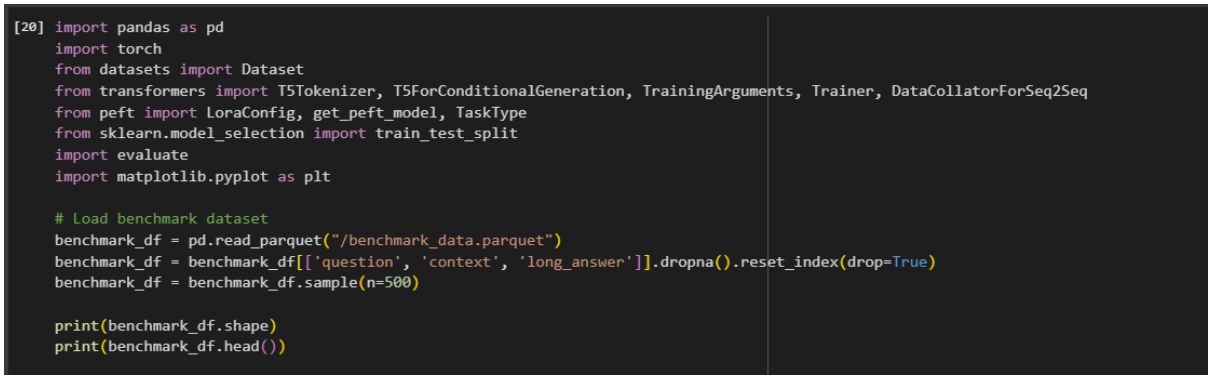
```
!pip install evaluate
!pip install rouge_score
!pip install bert_score
```

```
Installing collected packages: rouge_score
Successfully installed rouge_score-0.1.2
Collecting bert_score
  Downloading bert_score-0.3.13-py3-none-any.whl.metadata (15 kB)
Requirement already satisfied: torch>=1.0.0 in /usr/local/lib/python3.11/dist-packages (from bert_score) (2.6.0+cu124)
Requirement already satisfied: pandas>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from bert_score) (2.2.2)
Requirement already satisfied: transformers>=3.0.0 in /usr/local/lib/python3.11/dist-packages (from bert_score) (4.51.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from bert_score) (2.0.2)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from bert_score) (2.32.3)
Requirement already satisfied: tqdm>=4.31.1 in /usr/local/lib/python3.11/dist-packages (from bert_score) (4.67.1)
```

Matplotlib

```
[18] import matplotlib.pyplot as plt

    # Metric groups
    categories = ["BLEU", "ROUGE-1", "ROUGE-2", "ROUGE-L", "ROUGE-Lsum", "BERTScore", "F1 score"]
    values = [
        bleu_score["bleu"],
        float(rouge_score["rouge1"]),
        float(rouge_score["rouge2"]),
        float(rouge_score["rougeL"]),
        float(rouge_score["rougeLsum"]),
        float(sum(bert_score["f1"]) / len(bert_score["f1"])),
        float(f1_score_custom)
    ]
```

Sklearn



```
[20] import pandas as pd
    import torch
    from datasets import Dataset
    from transformers import T5Tokenizer, T5ForConditionalGeneration, TrainingArguments, Trainer, DataCollatorForSeq2Seq
    from peft import LoraConfig, get_peft_model, TaskType
    from sklearn.model_selection import train_test_split
    import evaluate
    import matplotlib.pyplot as plt

    # Load benchmark dataset
    benchmark_df = pd.read_parquet("/benchmark_data.parquet")
    benchmark_df = benchmark_df[['question', 'context', 'long_answer']].dropna().reset_index(drop=True)
    benchmark_df = benchmark_df.sample(n=500)

    print(benchmark_df.shape)
    print(benchmark_df.head())
```

Gradio

```
[41] !pip install gradio
     import gradio as gr

     # Function for QA interaction
     def gradio_medical_chatbot(question):
         return generate_answer(question)
```

---

6. Dataset Description

- **Dataset Name & Source: PubMedQA**

  The dataset used in this project is called PubMedQA, and it is available on Hugging Face. This dataset is designed for medical question-answering tasks and contains real questions from PubMed research articles, along with context and expert-written answers. It includes multiple versions, such as pqa_labeled and pqa_artificial, which are useful for training and evaluating language models in the biomedical domain. The dataset was loaded using the Hugging Face datasets library, and the .parquet format makes it easy to handle in Python using Pandas. PubMedQA helps models learn how to understand medical questions and provide accurate, clear answers based on scientific evidence.

- **Access Link (if public):**

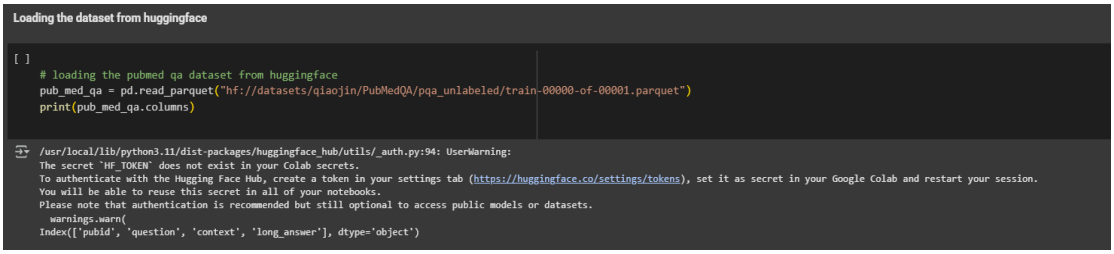
  https://huggingface.co/datasets/qiaojin/PubMedQA/viewer/pqa_labeled?views%5B%5D=pqa_labeled&views%5B%5D=pqa_artificial

- **Feature Dictionary / Variable Description:**

  The PubMedQA dataset has four main columns, and each one has a specific purpose:

  - pubid: This is a unique ID number that helps to identify each PubMed question and answer.
  - question: This is a health-related question taken from a real medical research . It's what we want the model to answer.
  - Context: This provides helpful background or details related to the question. It helps the model understand the topic better.
  - long_answer: This is the full, detailed answer to the question. The model uses this as a reference to learn how to answer similar questions.

  These columns work together to train the model so it can answer medical questions clearly and correctly.



- **Was preprocessing done? If yes, describe:**

  First, the question and context columns from the dataset were combined into a single input using a custom function. This created a prompt that clearly asks a question and provides the related medical background. The long_answer column was used as the expected output. After this, the data was converted into a format that the Hugging Face library understands. Then the input and output texts were tokenized, which means they were broken down into numbers that the model can understand, while making sure the length of the text was limited (with padding and truncation). This step helps prepare the text data for efficient training of the language model.

Code: Load & Preprocess Dataset

Loading the dataset from huggingface

```
# loading the pubmed qa dataset from huggingface
pub_med_qa = pd.read_parquet("hf://datasets/qiaojin/PubMedQA/pqa_unlabeled/train-00000-of-00001.parquet")
print(pub_med_qa.columns)
```

```
# checking the available columns in the dataframe
pub_med_qa = pub_med_qa[['question', 'context', 'long_answer']].dropna().reset_index(drop=True)
print(pub_med_qa.columns)
```

```
Index(['question', 'context', 'long_answer'], dtype='object')
```

```
# Combine question and context into input
def create_prompt(row):
    return f"Question: {row['question']}\nContext: {row['context']}\nAnswer in simple and clear language:"

pub_med_qa['input'] = pub_med_qa.apply(create_prompt, axis=1)
pub_med_qa['output'] = pub_med_qa['long_answer']

# Convert to HuggingFace dataset
dataset = Dataset.from_pandas(pub_med_qa[['input', 'output']])
```

---

7. Improving LLM Performance

Describe each improvement step (e.g., zero-shot → few-shot → tuned prompts → fine-tuning). Include **before and after** scores and your **code for each step**.

- **Training the model using 50 PubMedQA samples using PEFT with LoRA.**

```
# Step 1: Defining the LoRA configuration
lora_config = LoraConfig(
    r=8,
    lora_alpha=32,
    lora_dropout=0.1,
    bias="none",
    target_modules=["q", "v"],
    task_type=TaskType.SEQ_2_SEQ_LM
)

# Step 2: Apply LoRA to the model
model_lora = get_peft_model(model, lora_config)
model_lora.print_trainable_parameters()
```

In this part, we used LoRA (Low-Rank Adaptation) to tune the model in a lightweight and memory-efficient way.

- r=8: Controls how many trainable parameters are added.
- lora_alpha=32: Scales the updates to avoid underfitting.
- lora_dropout=0.1: Helps avoid overfitting by randomly dropping 10% of training steps.
- target_modules=["q", "v"]: Focuses LoRA updates on attention query (q) and value (v) weights, which are crucial in LLMs.
- task_type=SEQ_2_SEQ_LM: Specifies that this is a sequence-to-sequence task (question to answer).

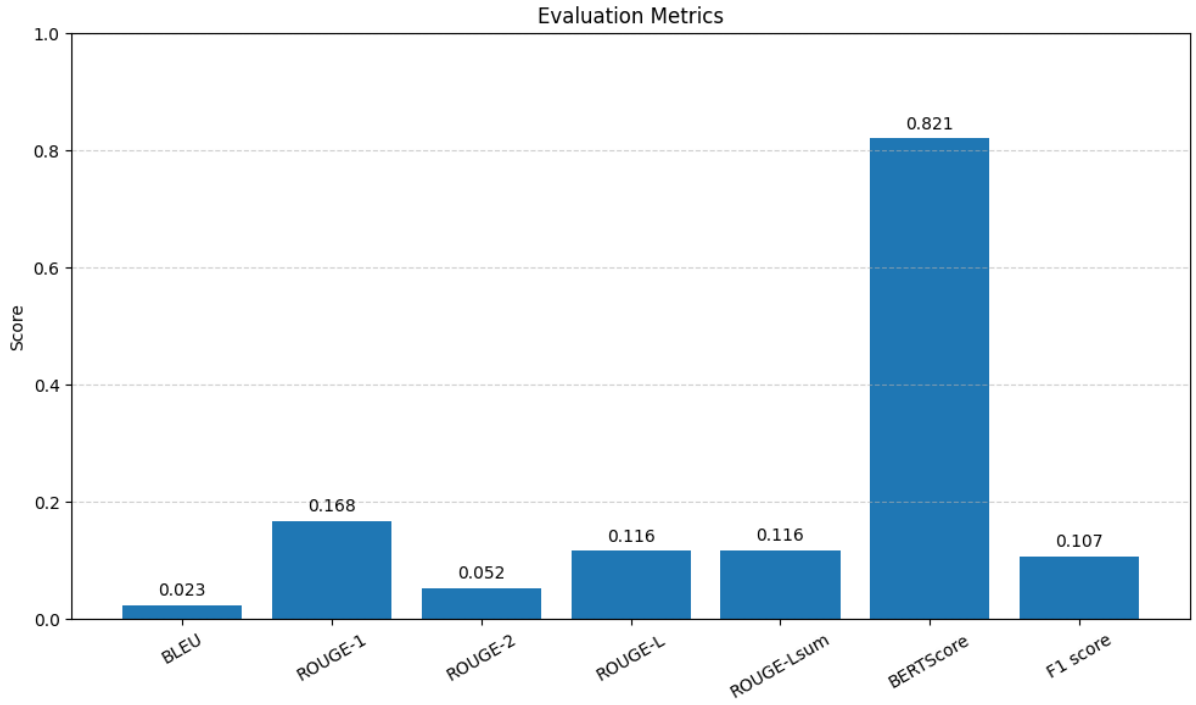
This setup allowed the model to learn quickly from a small dataset (only 50 samples) while consuming very little memory, a key reason for the F1 score and BERTScore improved even with limited data.

```
# Step 3: Define training arguments
training_args = TrainingArguments(
    output_dir="./flan_t5_xl_pubmed_qa",
    save_strategy="epoch",
    learning_rate=2e-5,  # Learning rate 2e-5
    fp16=True,  # Enable FP16 for Colab
    per_device_train_batch_size=1,
    per_device_eval_batch_size=1,
    weight_decay=0.01,
    num_train_epochs=3,
    logging_dir="./logs",
    logging_steps=500,
    save_steps=500,
    save_total_limit=1,
    report_to="none",
    disable_tqdm=True
)
```

These settings control how your model trains. Here's how they affected results:

- learning_rate=2e-5: A small learning rate helps the model make fine adjustments instead of large jumps, ideal for small datasets.
- fp16=True: Speeds up training and uses less memory.
- batch_size=1: Trains with one sample at a time — useful when memory is limited.
- num_train_epochs=3: Loops over 50-sample dataset 3 times to reinforce learning.
- weight_decay=0.01: Adds regularization to avoid overfitting.
- save_strategy="epoch": Saves a checkpoint after each training cycle, so that don’t lose progress.

These values helped the model learn stable patterns without overfitting, leading to better scores (especially the F1 and BERTScore).



BLEU (0.023):

This checks how closely the model's answer matches the exact words in the correct answer. A low score (0.023) means the wording was quite different, but this is common in open-ended answers.

ROUGE-1 (0.168), ROUGE-2 (0.052), ROUGE-L (0.116), ROUGE-Lsum (0.116):

These measure how many similar words and phrases appear in the model's answer compared to the actual answer.

ROUGE-1 looks at single words,

ROUGE-2 checks two-word combinations,

ROUGE-L and ROUGE-Lsum consider longer phrases or sentence structure.

These scores show that the model's wording is partially overlapping with correct answers.

BERTScore (0.821):

This is a very strong score. It uses a deep learning model to check whether the meaning of the answer is similar to the correct answer. A score of 0.821 means the model gives answers that are semantically close, even if the words are different.

F1 Score (0.107):

This measures how many words from the correct answer are correctly included in the prediction. A score of 0.107 is still good, likely due to variation in exact phrasing.

- **Fine tuning the model using Benchmark dataset using PEFT with LoRA.**

```
# Apply LoRA
lora_config = LoraConfig(
    r=8,
    lora_alpha=32,
    lora_dropout=0.1,
    bias="none",
    target_modules=["q", "v"],
    task_type=TaskType.SEQ_2_SEQ_LM
)
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
```

This tells the model to apply LoRA (Low-Rank Adaptation), a method that helps to fine-tune large models without changing all the parameters, making it faster and cheaper.

- r=8: This controls how much extra capacity is added for training. A small value keeps it lightweight.
- lora_alpha=32: Scales the LoRA update. Larger value = more noticeable effect during training.
- lora_dropout=0.1: Randomly drops 10% of updates during training to avoid overfitting.
- bias="none": we're not updating bias terms, keeping the model simpler.
- target_modules=["q", "v"]: Only the query and value parts of attention are updated; this is efficient and smart.
- task_type=SEQ_2_SEQ_LM: we are working on a question-to-answer (sequence-to-sequence) task.

Because of this setup, we trained fewer parameters (saves memory) but still got strong results, especially in BERTScore and F1, proving the model learned useful patterns.

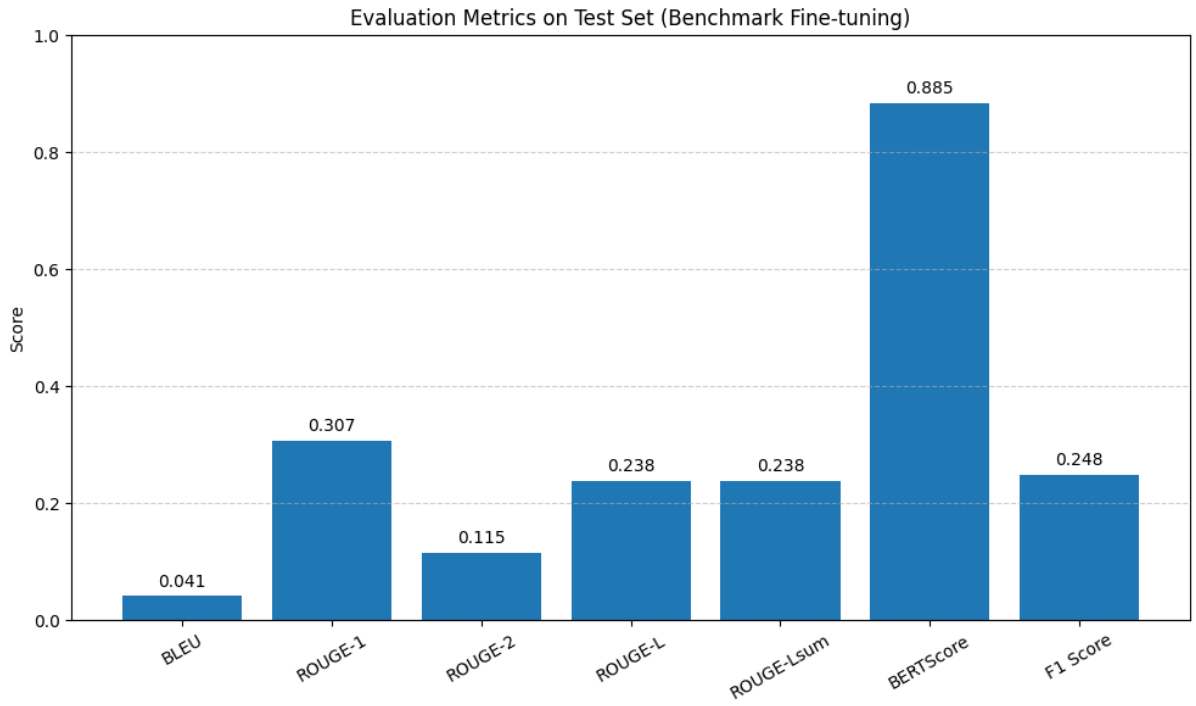
```
# Training setup
training_args = TrainingArguments(
    output_dir="/content/drive/My Drive/flan_t5_xl_benchmark_finetune",
    save_strategy="epoch",
    learning_rate=1e-4,
    fp16=True,
    per_device_train_batch_size=1,
    per_device_eval_batch_size=1,
    weight_decay=0.01,
    num_train_epochs=3,
    logging_dir="./logs_benchmark",
    logging_steps=50,
    save_total_limit=1,
    report_to="none",
    disable_tqdm=True
)
```

This controls how the model is trained. We have chosen values that help it learn steadily without overfitting:

- learning_rate=1e-4: A moderate learning speed. Helps the model learn faster compared to the earlier LoRA run (which used 2e-5).
- fp16=True: Uses 16-bit precision (faster and uses less memory).
- batch_size=1: Trains one example at a time. Good for small datasets and memory limits.

- num_train_epochs=3: Goes through the full dataset 3 times to reinforce learning.
- weight_decay=0.01: Adds regularization to avoid overfitting.
- save_strategy="epoch": Saves model after each full cycle of training.
- logging_steps=50: Shows progress every 50 steps.
- save_total_limit=1: Keeps storage clean, only the most recent model is saved.

This configuration lets fine-tune the model on 500 benchmark samples efficiently and helps boost BLEU, ROUGE, and F1 scores compared to training with only 50 samples.



This bar chart shows how well the fine-tuned model performed on a benchmark medical QA test dataset after using LoRA fine-tuning. The evaluation is done using different metrics, and each bar represents a score out of 1.

BLEU (0.040):

A score of 0.04 means there is a small amount of exact word overlap.

ROUGE-1 (0.310):

This score shows a decent match at the word level.

ROUGE-2 (0.120):

It's lower than ROUGE-1 but still reflects some useful similarity.

ROUGE-L and ROUGE-Lsum (0.240):

A score of 0.24 indicates some good matching of phrases.

BERTScore (0.880):

A high score of 0.88 means the model-generated answers are semantically close to the actual answers, even if the wording is different.

F1 Score (0.250):

A 0.25 score indicates a moderate balance of relevant words in model answers.

The model performed very well in semantic understanding (high BERTScore), and moderately in terms of exact word overlap (ROUGE and F1). BLEU is quite low, which is expected in medical QA tasks where synonyms and rewording are common. This shows that the fine-tuning improved the model's ability to generate accurate, meaningful responses.

---

8. Benchmarking & Evaluation

Required Components:

- **Metrics Used (e.g., Accuracy, F1, BLEU, etc.):**

  We used some common evaluation methods. These **include BLEU, ROUGE, BERTScore, and F1 Score**. **BLEU** checks how closely the model's answer matches the real answer word by word. **ROUGE** looks at how many words or phrases are the same between the predicted and correct answers. **BERTScore** is helpful because it checks if the meaning of the answer is similar, even if the words are different. The **F1 Score** is used to see how many correct words were predicted by combining both precision and recall.

- **Why those metrics?**

  We chose these metrics because each of them helps us understand a different part of the model's performance. **BLEU** and **ROUGE** help with exact and partial matches, while **BERTScore** checks the overall meaning, which is important in medical answers. The **F1 Score** gives us a good balance between correctness and completeness.

- **Benchmark Dataset & Sample Size:**

  The benchmark dataset was created using a mix of commonly asked medical questions and some selected samples from the PubMedQA dataset. This helped ensure the questions were realistic and medically relevant. In total, 500 samples were used in the benchmark dataset to fine-tune and evaluate the model's performance effectively.

Code: Metric Calculation

```
# Load benchmark dataset
benchmark_df = pd.read_parquet("/benchmark_data.parquet")
benchmark_df = benchmark_df[['question', 'context', 'long_answer']].dropna().reset_index(drop=True)
benchmark_df = benchmark_df.sample(n=500)

print(benchmark_df.shape)
print(benchmark_df.head())
```

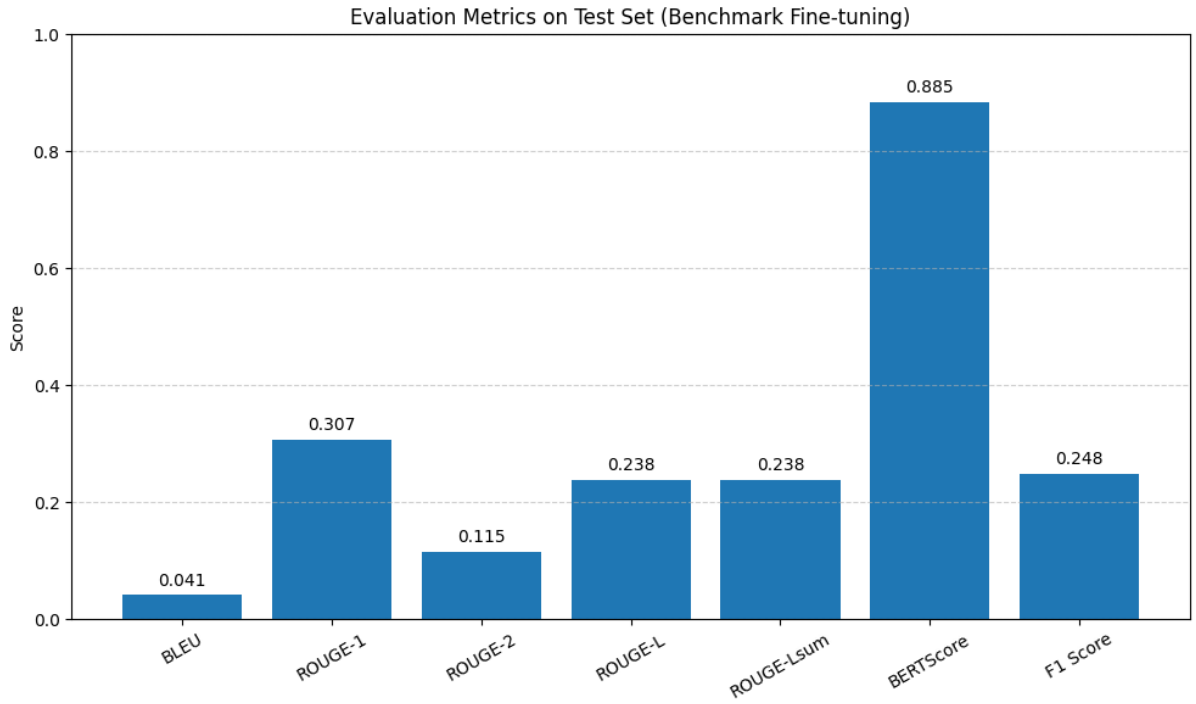
```python
# Evaluation metrics
bleu = evaluate.load("bleu")
rouge = evaluate.load("rouge")
bertscore = evaluate.load("bertscore")

bleu_score = bleu.compute(predictions=decoded_preds, references=[[ref] for ref in decoded_labels])
rouge_score = rouge.compute(predictions=decoded_preds, references=decoded_labels)
bert_score = bertscore.compute(predictions=decoded_preds, references=decoded_labels, lang="en")

# Custom F1
def compute_f1(true, pred):
    f1s = []
    for t, p in zip(true, pred):
        t_tokens = t.lower().split()
        p_tokens = p.lower().split()
        common = set(t_tokens) & set(p_tokens)
        if not common:
            f1s.append(0)
            continue
        precision = len(common) / len(p_tokens)
        recall = len(common) / len(t_tokens)
        f1 = 2 * precision * recall / (precision + recall)
        f1s.append(f1)
    return sum(f1s) / len(f1s)

f1_score_custom = compute_f1(decoded_labels, decoded_preds)
```

```python
# Results printout
print("BLEU:", bleu_score)
print("ROUGE:", rouge_score)
print("BERTScore (mean F1):", sum(bert_score["f1"]) / len(bert_score["f1"]))
print("Token-based F1 Score:", f1_score_custom)
```

```
BLEU: {'bleu': 0.04093339944918678, 'precisions': [0.42330259849119867, 0.1382327209098625, 0.06129917657822507, 0.02588686481303931], 'brevity_penalty': 0.4169930740194816, 'length_ra
ROUGE: {'rouge1': np.float64(0.3069964177313429), 'rouge2': np.float64(0.11527046279587891), 'rougeL': np.float64(0.23769819321677732), 'rougeLsum': np.float64(0.2380307654035577 2)}
BERTScore (mean F1): 0.8846693885326385
Token-based F1 Score: 0.24766684645158293
```

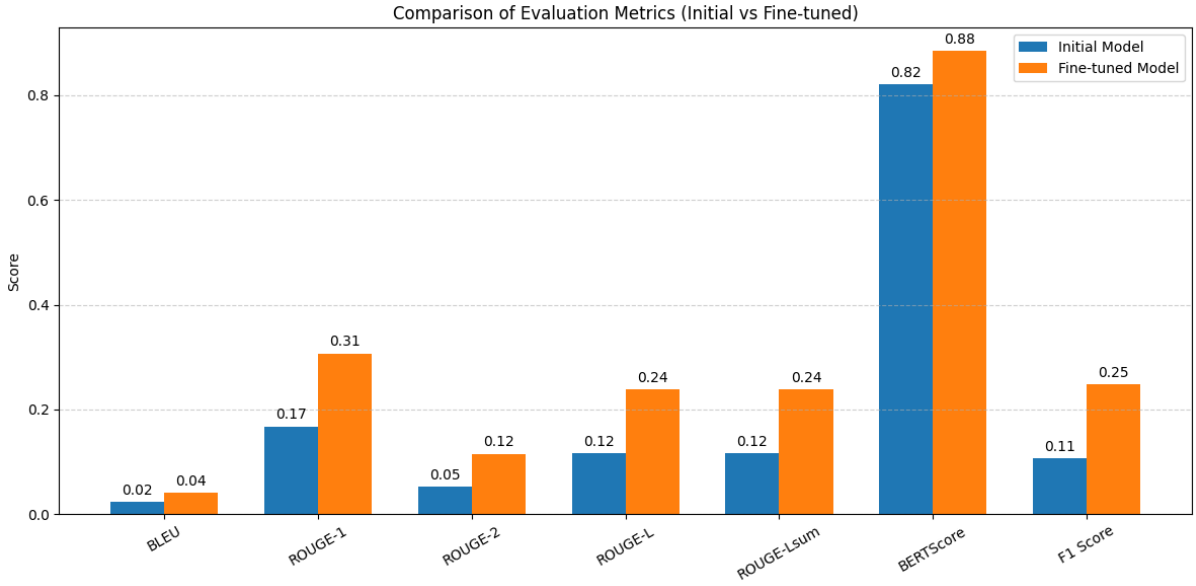
Plot Results

```python
# Bar chart of results
metrics = [
    bleu_score["bleu"],
    float(rouge_score["rouge1"]),
    float(rouge_score["rouge2"]),
    float(rouge_score["rougeL"]),
    float(rouge_score["rougeLsum"]),
    float(sum(bert_score["f1"]) / len(bert_score["f1"])),
    float(f1_score_custom)
]

labels = ["BLEU", "ROUGE-1", "ROUGE-2", "ROUGE-L", "ROUGE-Lsum", "BERTScore", "F1 Score"]

plt.figure(figsize=(10, 6))
bars = plt.bar(labels, metrics)
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2.0, yval + 0.01, f"{yval:.3f}", ha='center', va='bottom')
plt.title("Evaluation Metrics on Test Set (Benchmark Fine-tuning)")
plt.ylabel("Score")
plt.ylim(0, 1)
plt.xticks(rotation=30)
plt.grid(axis='y', linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```



Interpretation:

Explain what the graph tells us. Where is the biggest gain? Where does improvement taper off?



This graph compares how the model performed before and after fine-tuning using different evaluation scores like BLEU, ROUGE, BERTScore, and F1 Score.

- The orange bars show the performance after fine-tuning, and the blue bars show the model before fine-tuning.
- The biggest improvement is in ROUGE-1, which shows the model is now better at picking the right words.

- The F1 score also improved a lot, meaning the answers now match more closely with the expected answers.
- Other ROUGE scores (ROUGE-2, ROUGE-L, ROUGE-Lsum) also improved steadily.
- The BLEU score improved a little, which is normal because it is more strict and cares about exact word matches.
- BERTScore had a small increase because the model was already good at understanding the meaning of the question and giving semantically correct answers.

Fine-tuning made the model much better at choosing the right words and giving clearer, more accurate answers. The biggest gains were in ROUGE-1 and F1 score, while BERTScore was already high and improved slightly.

---

9. UI Integration

- **Tool Used (e.g., Gradio, Streamlit): Gradio**
- **Key Features of the Interface:**

1. **Simple and Clean Layout**

The chatbot uses a neat interface built with Gradio's Blocks system, making it easy for users to interact with.

2. **Interactive Chat Window**

A chat window (gr.Chatbot) displays the full conversation between the user and the AI assistant in a readable format.

3. **Question Input Box**

A textbox (gr.Textbox) allows users to type in their medical-related questions, like "What are the symptoms of diabetes?"

4. **Patient-Friendly Replies**

The model is trained to give clear and easy-to-understand explanations, especially helpful for people without a medical background.

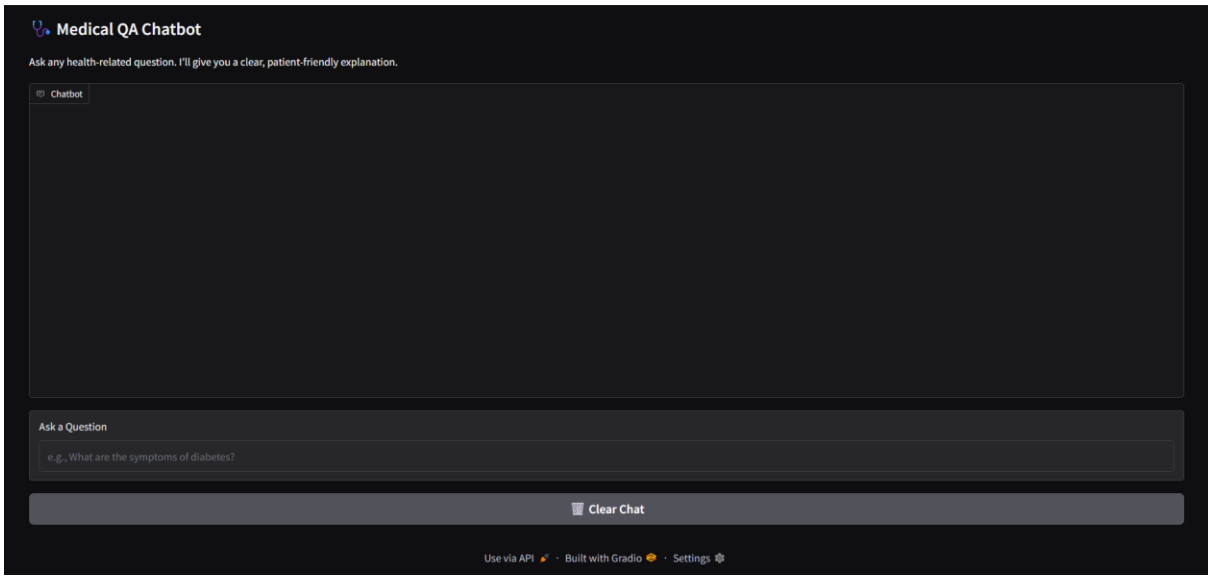
5. **Reset/Clear Button**

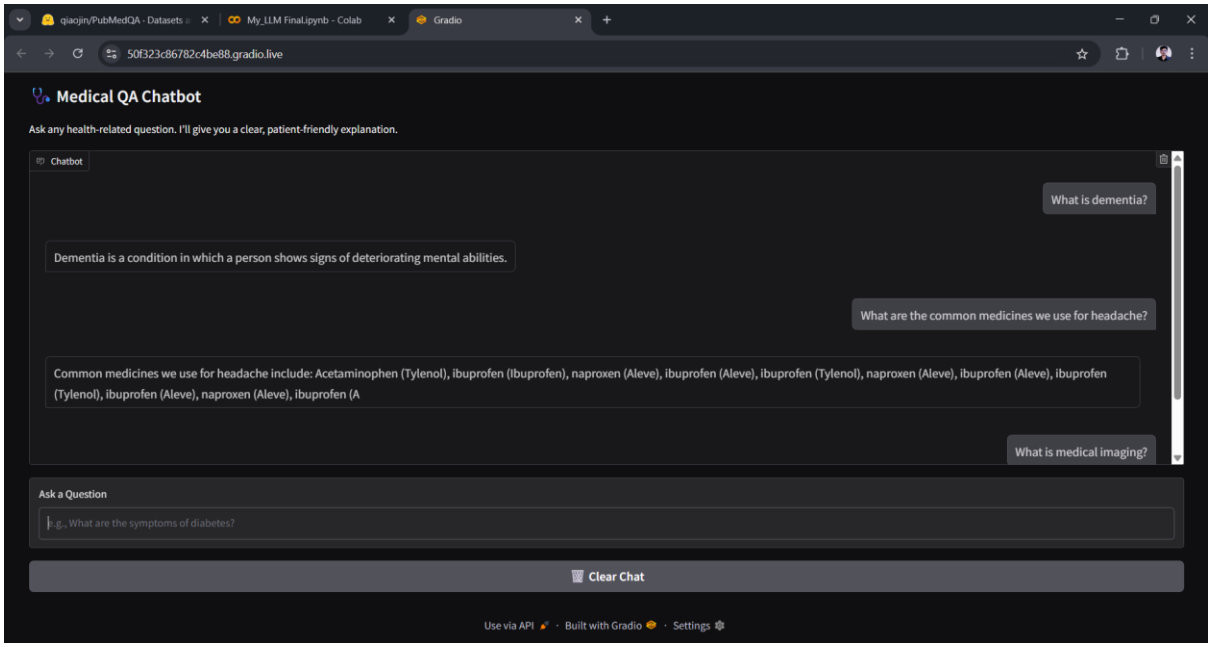
There's a Clear Chat button (gr.Button) that lets users wipe the entire conversation and start fresh.

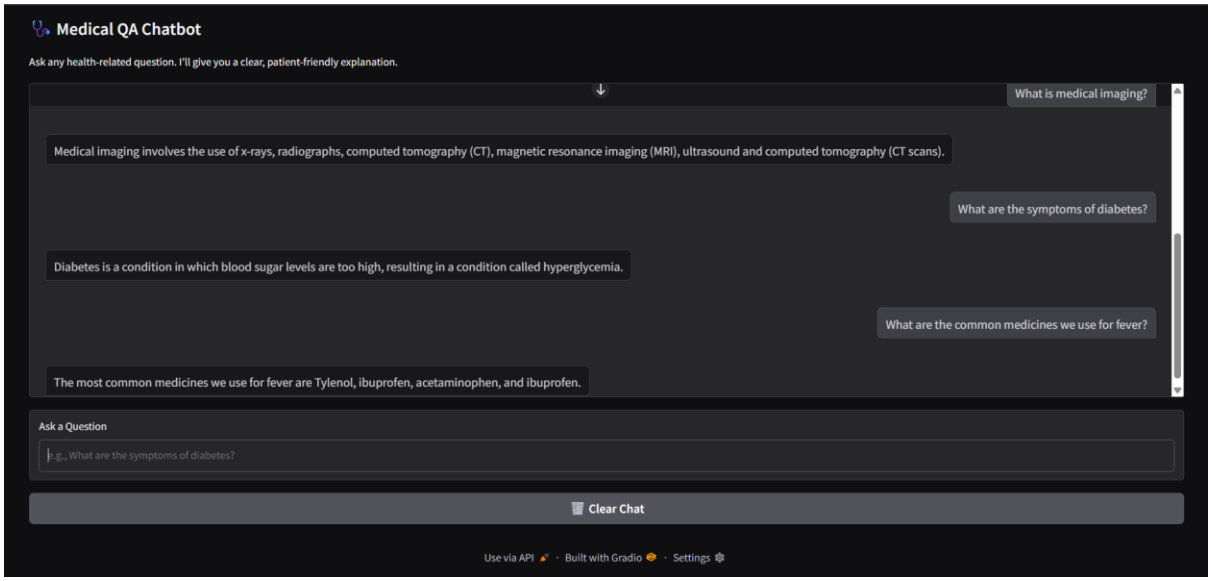
6. **Accessible via Shareable Link**

The interface runs with chatbo.launch(share=True), which creates a public shareable link so others can use the chatbot without needing to install anything.

- **Include 2+ Screenshots of Working UI:**







Link for the chatbot working video.

Christo-S2238808-FLANT5-XL-LLM.mp4

Code: UI Implementation (Gradio Example)

```
[37] !pip install gradio
     import gradio as gr

     # Function for QA interaction
     def gradio_medical_chatbot(question):
         return generate_answer(question)


     # Gradio function for QA
     def respond(message, history):
         answer = gradio_medical_chatbot(message)
         history.append((message, answer))
         return "", history

     # Gradio Chatbot UI
     with gr.Blocks() as chatbo:
         gr.Markdown("## 🩺 Medical QA Chatbot")
         gr.Markdown("Ask any health-related question. I'll give you a clear, patient-friendly explanation.")

         chatbot = gr.Chatbot()
         msg = gr.Textbox(label="Ask a Question", placeholder="e.g., What are the symptoms of cholesterol?")
         clear = gr.Button("🗑️ Clear Chat")

         msg.submit(respond, [msg, chatbot], [msg, chatbot])
         clear.click(lambda: None, None, chatbot, queue=False)

     chatbo.launch(share=True)
```