Fine-Tuning LLMs for Enterprise Applications

---

1. Student Information

- **Student Name:** Zaeem Rizan
- **Student ID:** s223134187
- **Date Submitted:** 11/05/2025

---

2. Project Introduction

- **Title of the Project:** Personalized Healthcare QA System (Chatbot)
- **What is the project about?**

  This project evaluates how well the Mistral 7B Large Language Model answers questions, both before and after a fine-tuning process. I selected the Mistral 7B model because it showed strong ability in complex tasks, like summarizing long articles and handling questions that need more than basic fact recall. The main method is fine-tuning this model. I adjusted its parameters to improve answer accuracy. For this, I used the PubMedQA dataset, a large collection of medical question and answer data. This dataset trained the model for health-related questions. I then tested its performance on similar questions.
- **Why is this project important or useful?**

  Specialized AI tools, like the one in this project, meet a growing need in healthcare. A fine-tuned QA chatbot, for example, can act as a first contact point in clinics. It can help with basic patient questions before they see a doctor. Training the model with more detailed and specific medical data should improve its ability to give accurate and relevant health information. This system is not a replacement for professional medical practitioners. However, it can be a useful first resource for people seeking health information.

---

3. API Token Setup

This section explains how I obtained and utilized an API token from Hugging Face for Large Language Model (LLM) access.

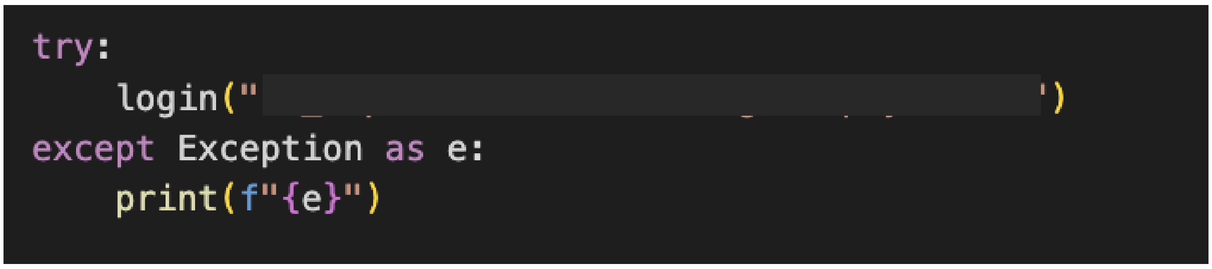
Provider Identification:

- I selected Hugging Face as the API provider for this project. The project notebook uses the Hugging Face login function to authenticate. Hugging Face offers a wide range of language models and tools for natural language processing tasks.

Token Generation Process: I acquired the Hugging Face API token by following these standard steps:

- Step 1: I created a user account on the Hugging Face website (huggingface.co).
- Step 2: I accessed my user profile settings and located the "Access Tokens" area.
- Step 3: I initiated the creation of a new token.
- Step 4: I assigned a name to the token and configured its permissions, such as read access. I then generated the token.
- Step 5: I copied the generated token and stored it in a secure location for my project's use.

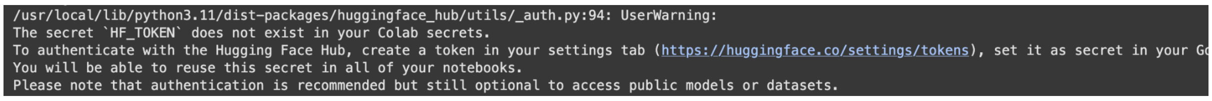
Token Implementation: The project incorporates the token through the following Python commands:



```
try:
    login("[redacted]')
except Exception as e:
    print(f"{e}")
```

I imported the HuggingFace API using the above.

Screenshot or terminal output of successful loading of API:



```
/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://huggingface.co/settings/tokens), set it as secret in your Go
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 or datasets.
```

Although it is recommended to use secure methods for loading tokens, like using environment variables. For this project, I used the token directly within the `login()` function. I chose this approach because I was the sole developer and did not share the project code.

---

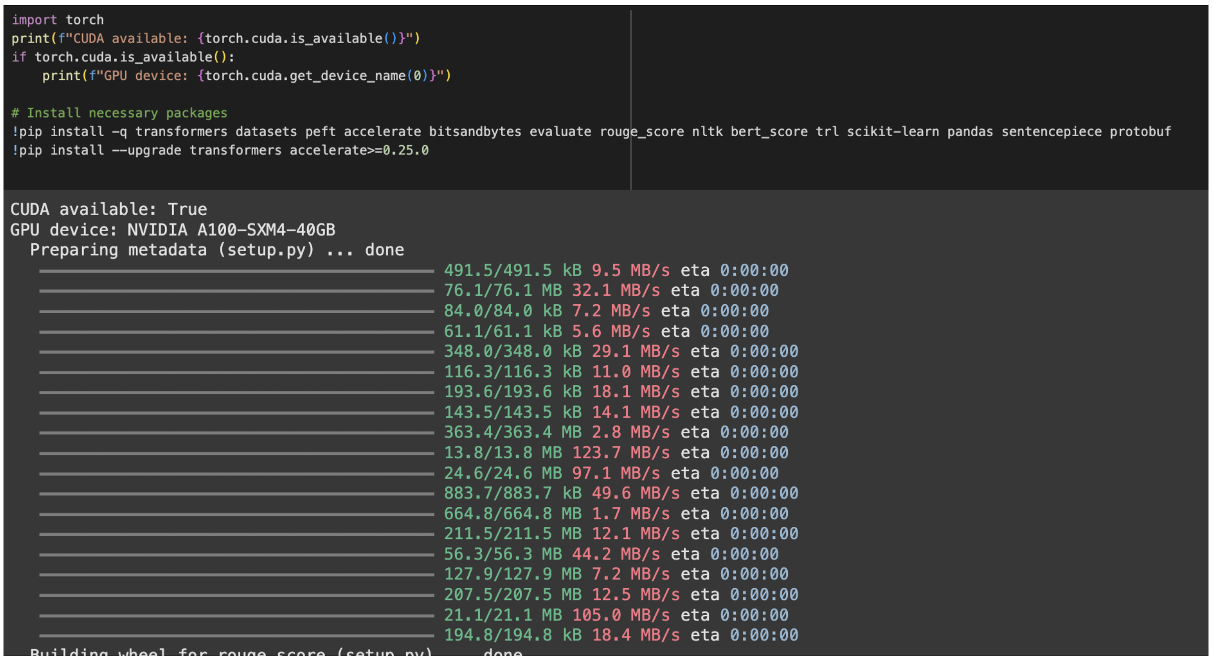
4. Environment Setup

Development Platform

I selected Google Colab for this project's development and execution. My local machine lacked a GPU with sufficient processing power for training the language model. Google Colab provides accessible cloud-based GPU resources. I specifically used the Google Colab Pro subscription to ensure adequate resources for the model training tasks. Other cloud services, such as Amazon Sagemaker, provide similar capabilities.

Tools Used

The project used an NVIDIA A100 GPU, which Google Colab Pro provided and Python 3.11



```python
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-learn pandas sentencepiece protobuf
!pip install --upgrade transformers accelerate>=0.25.0
```

```
CUDA available: True
GPU device: NVIDIA A100-SXM4-40GB
  Preparing metadata (setup.py) ... done
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 491.5/491.5 kB 9.5 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 76.1/76.1 MB 32.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.0/84.0 kB 7.2 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 61.1/61.1 kB 5.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 348.0/348.0 kB 29.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 11.0 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 193.6/193.6 kB 18.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 143.5/143.5 kB 14.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 2.8 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 123.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 97.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 49.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 1.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 12.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 44.2 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 7.2 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 12.5 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 105.0 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 194.8/194.8 kB 18.4 MB/s eta 0:00:00
  Building wheel for rouge_score (setup.py) ... done
```

---

5. LLM Setup

Model

I chose the Mistral 7B model for this project. I found its performance impressive, particularly its ability to summarize large articles and handle queries that require more than simple fact retrieval. I also found it to be effective at following instructions.

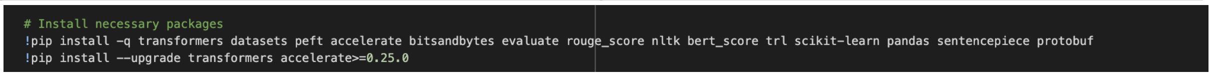
Provider

Hugging Face. The model and associated tools were accessed via the Hugging Face API and libraries.

Key Libraries & Dependencies

torch	This library provides tensor computation and is essential for building and training neural networks
transformers	I used it to download, load, and work with the pre-trained Mistral 7B model and its tokenizer.
datasets	Another Hugging Face library I used to load and process the PubMedQA dataset.

peft	This Hugging Face library allowed for efficient fine-tuning of the large Mistral 7B model
accelerate	this library helps run training scripts on different hardware when I tested with running on T4, CPU and A100GPUs
bitsandbytes	I used this library for model quantization. It helped load the Mistral 7B model using 4-bit precision
evaluate	I used to load and calculate standard performance metrics such as ROUGE and BERTScore.
nltk	I used this library for text processing. Specifically, the punkt tokenizer was downloaded for sentence segmentation
bert_score	This library provided the BERTScore metric, which measures the similarity between the model's generated answers and reference answers based on token embeddings.



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

The libraries were installed with the above and other libraries were installed in their respective cells when I encountered the need to use them.

---

6. Dataset Description

Dataset Name & Source

The project uses the PubMedQA dataset. Specifically, I used the pqa_labeled subset, which is available on Hugging Face Datasets under the identifier qiaojin/PubMedQA. This subset contains 1,000 labeled examples.

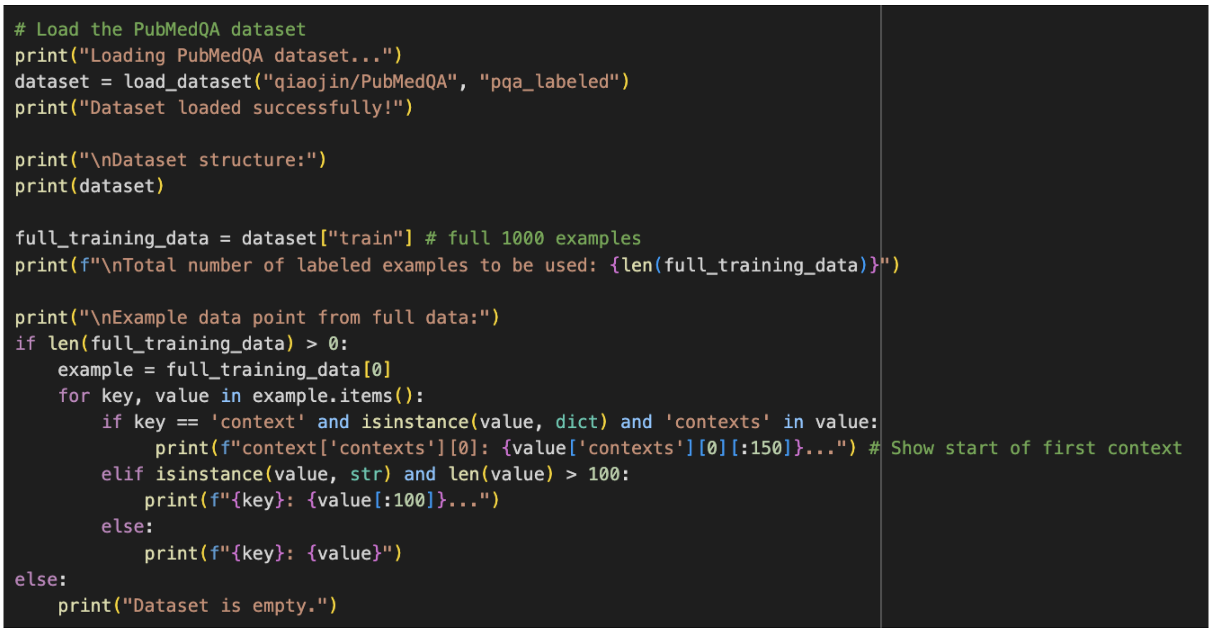
Access Link

https://huggingface.co/datasets/qiaojin/PubMedQA/

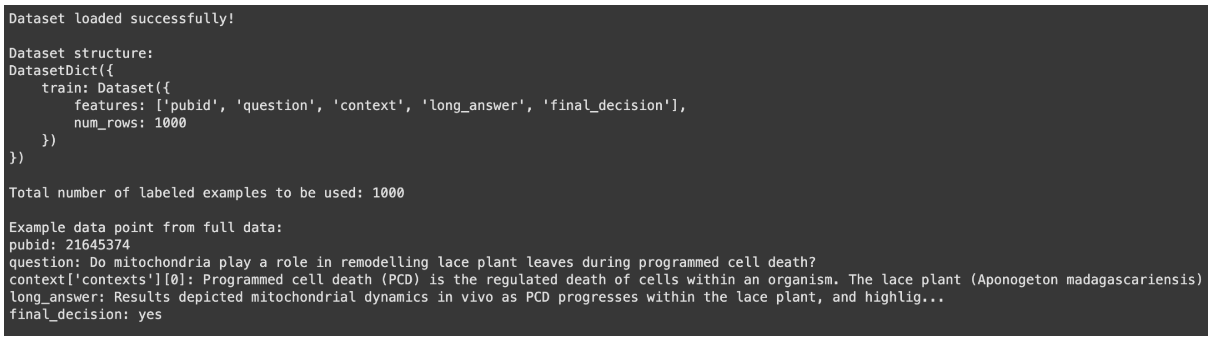
Feature Dictionary / Variable Description

The pqa_labeled subset of the PubMedQA dataset includes the following features for each example:

- **pubid:** An integer (int32) identifier for the publication from which the question and answer are derived.
- **question:** A string containing the medical question.
- **context:** A sequence or structure containing text passages from the medical article. These passages provide the context needed to answer the question.
- **long_answer:** A string that provides a detailed, free-form answer based on the context.
- **final_decision:** A concise answer to the question, typically "yes", "no", or "maybe".



```python
# 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.")
```



```
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 cell death?
context['contexts'][0]: Programmed cell death (PCD) is the regulated death of cells within an organism. The lace plant (Aponogeton madagascariensis)
long_answer: Results depicted mitochondrial dynamics in vivo as PCD progresses within the lace plant, and highlig...
final_decision: yes
```

Preprocessing the Dataset

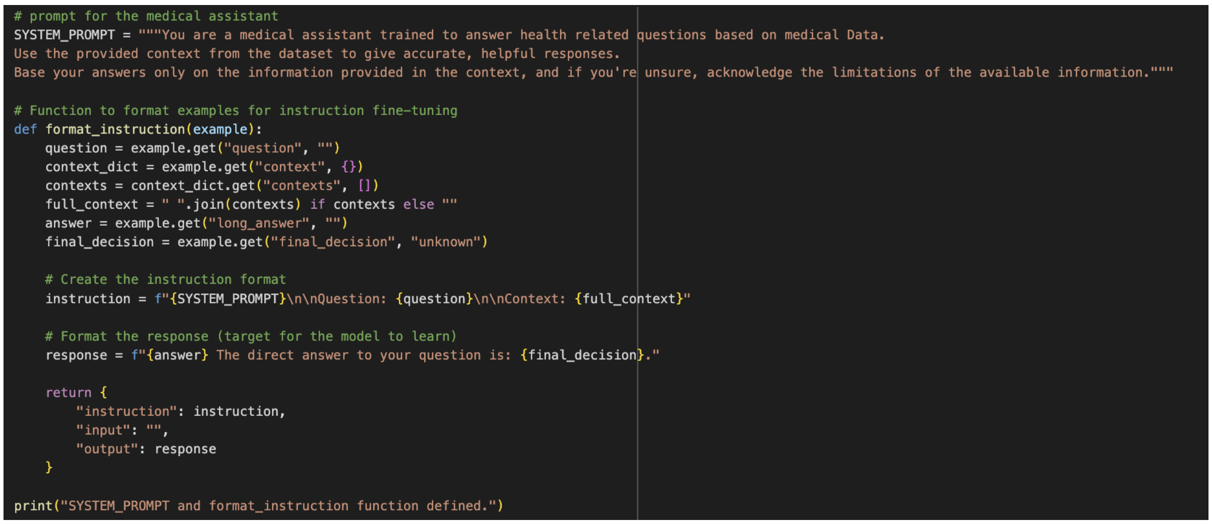
I performed preprocessing on the dataset. The main goal of this preprocessing was to transform the raw data into an instruction-response format. This format is suitable for fine-tuning the language model.

This transformation involved several steps:

- I defined a system prompt to guide the model's behavior, instructing it to act as a medical assistant.

- For each data example, I combined this system prompt, the original question, and the context (joining all context passages into a single text block) to create a complete input prompt for the model.
- I formatted the target output for the model by combining the long_answer and the final_decision into a single response string.

This structured prompt-response pair was then used to fine-tune the model.



```python
# 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 of the available information."""

# 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.")
```

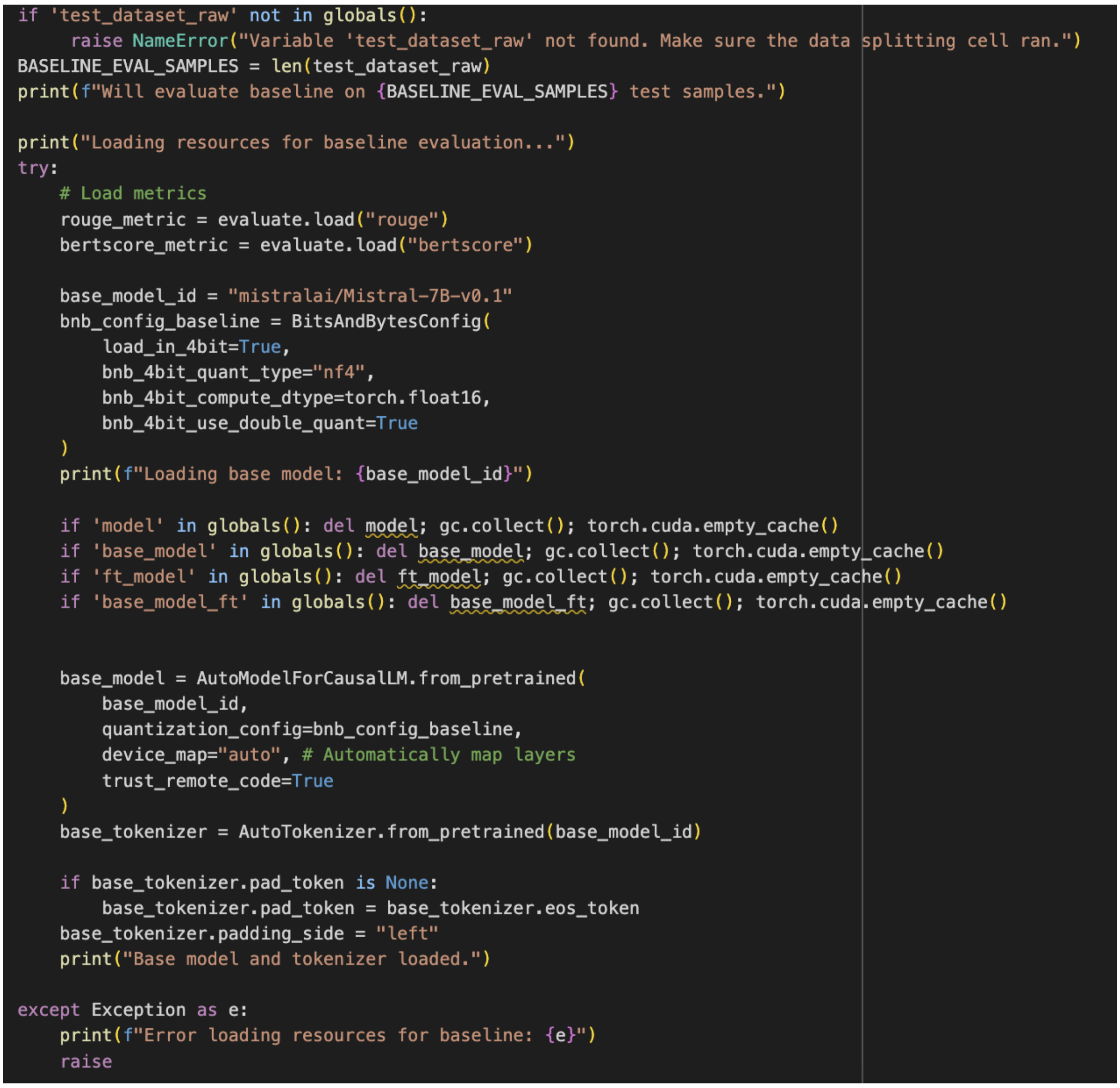
---

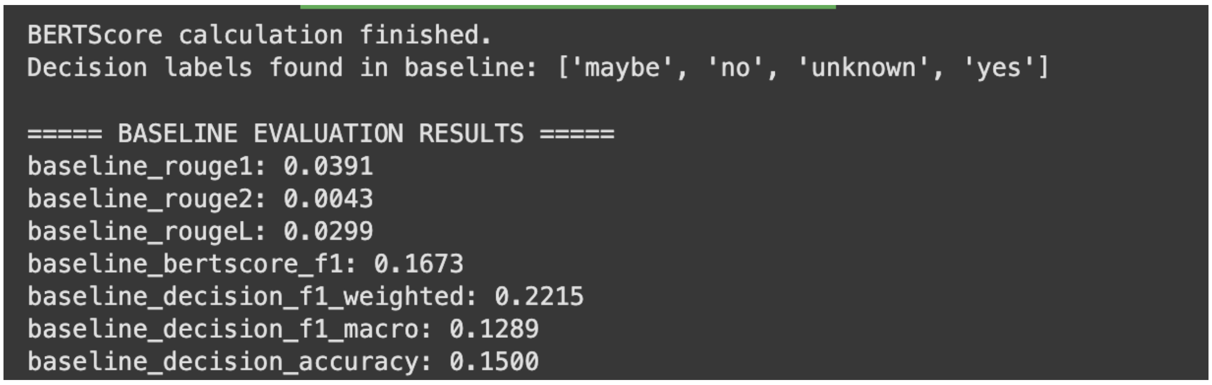
7. Improving LLM Performance

Baseline Performance Evaluation

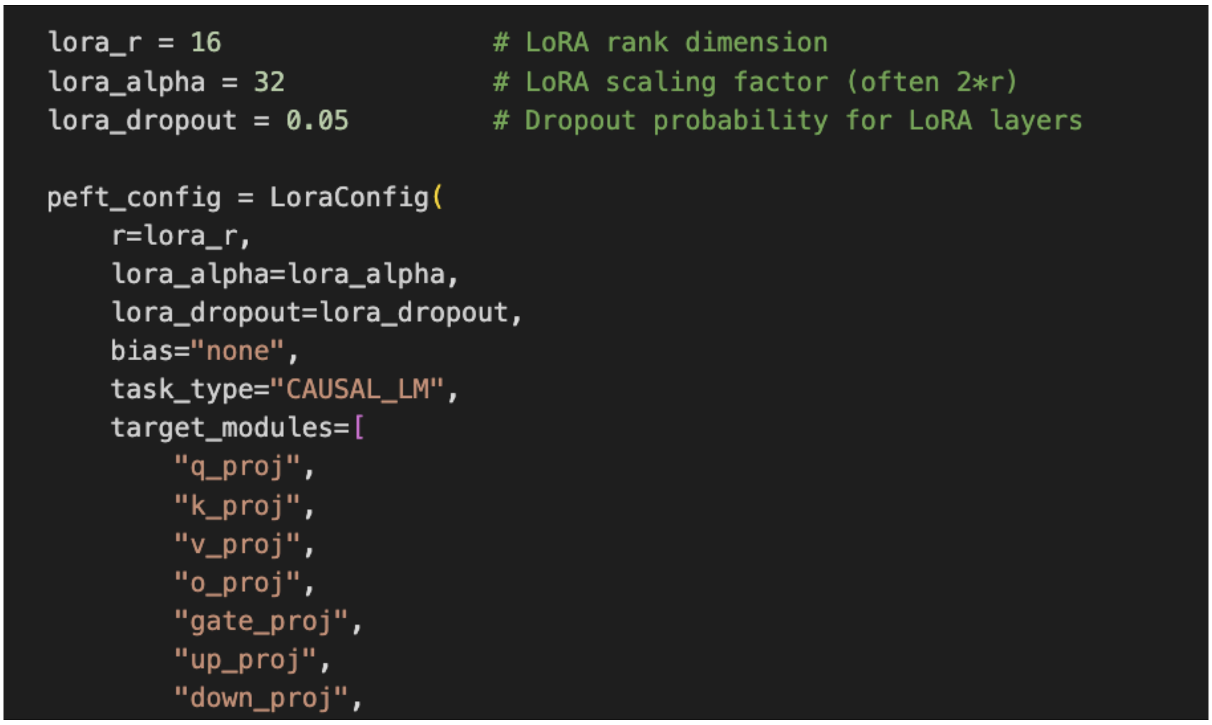
First, I established a baseline. I evaluated the original, pre-trained mistralai/Mistral-7B-v0.1 model on the test set (100 samples from PubMedQA) without any fine-tuning. This step measured the model's out-of-the-box capabilities for the question-answering task on this specific dataset. The model generated answers based on the provided context and question.

Key metrics such as ROUGE scores, BERTScore F1, and decision accuracy (for "yes/no/maybe" answers) were recorded. For instance, the baseline BERTScore F1 was 0.1673 and Decision Accuracy was 0.1500.


```python
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 '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
```


```
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
baseline_decision_accuracy: 0.1500
```

Next, I fine-tuned the Mistral 7B model using the QLoRA technique. This method allows efficient fine-tuning of large language models by quantizing the pre-trained model to 4-bit precision and then training small, additional "adapter" layers. I used the pqa_labeled training data (900 examples after a 90/10 split of the original 1000 examples) formatted into an instruction-response style for this process. The model trained for 4 epochs.

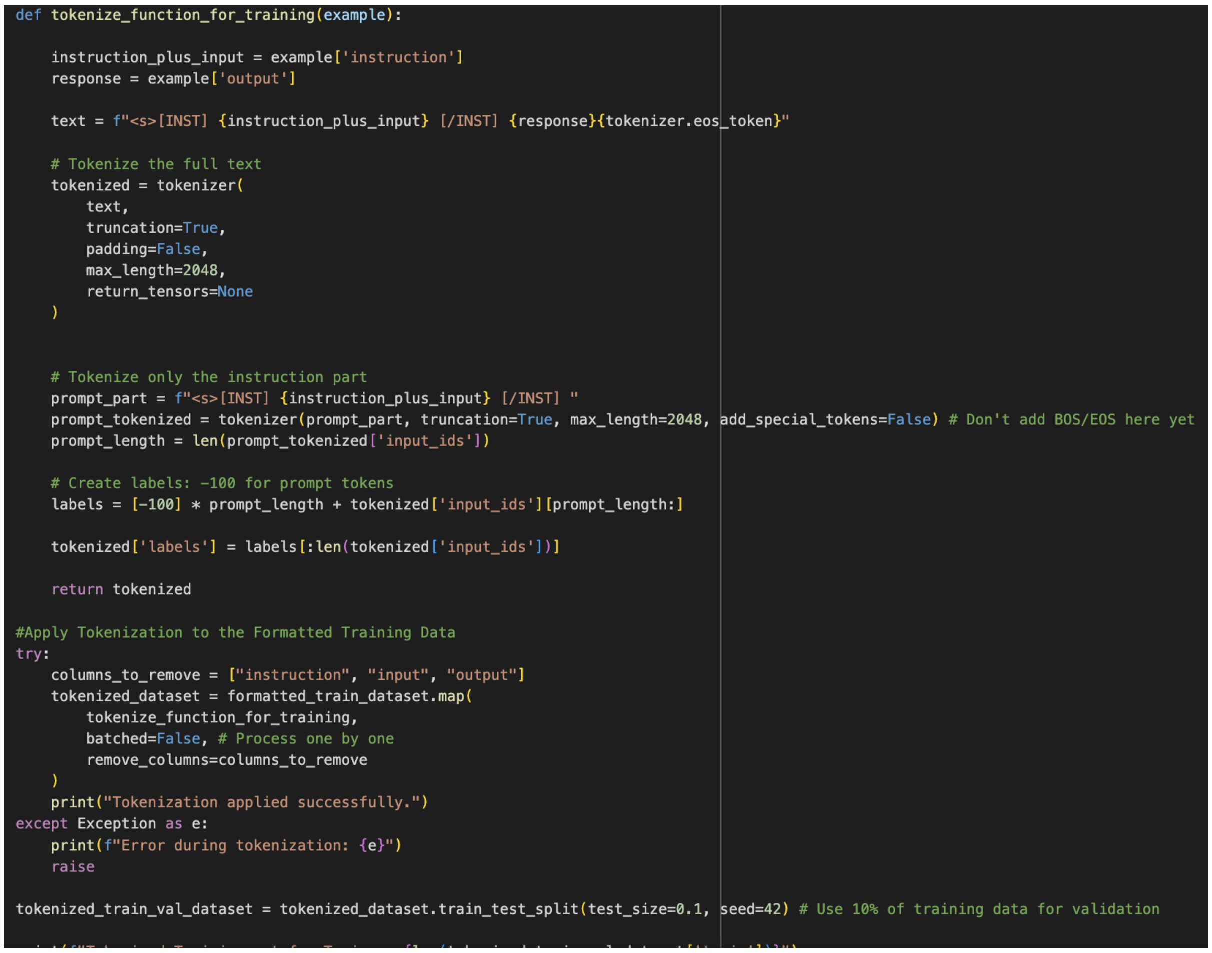


```
lora_r = 16                 # LoRA rank dimension
lora_alpha = 32             # LoRA scaling factor (often 2*r)
lora_dropout = 0.05         # Dropout probability for LoRA layers

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",
```

r	This sets the rank of the LoRA matrices to 16. A lower rank means fewer trainable parameters.
lora_dropout	This sets the dropout rate for the LoRA layers to 0.05.
bias	I set this to "none", so bias parameters in the LoRA layers are not trained.
task_type	I specified "CAUSAL_LM" because Mistral 7B is a causal language model (predicts the next token).
target_modules	This lists the specific linear layers within the Mistral 7B model where I applied the LoRA adapters.

Tokenization for Training

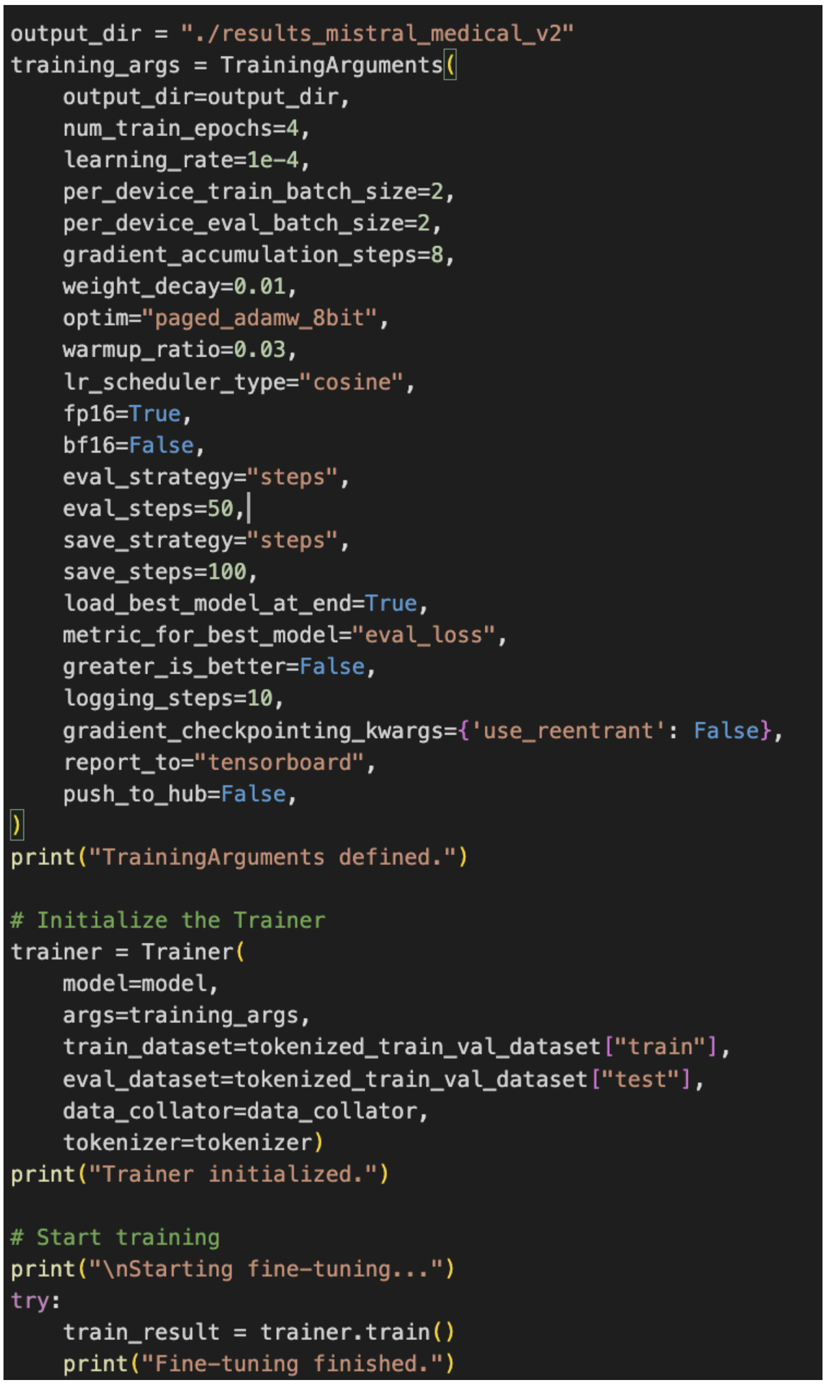
I tokenized the preprocessed instruction-response pairs. The tokenizer converts text into a numerical format the model understands. For training, I also created labels that the model uses to learn.



```python
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 BOS/EOS here yet
    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 for validation
```

This function ensures the model learns to generate the response part after seeing the instruction part, by setting the labels for the instruction tokens to -100, which the loss function ignores

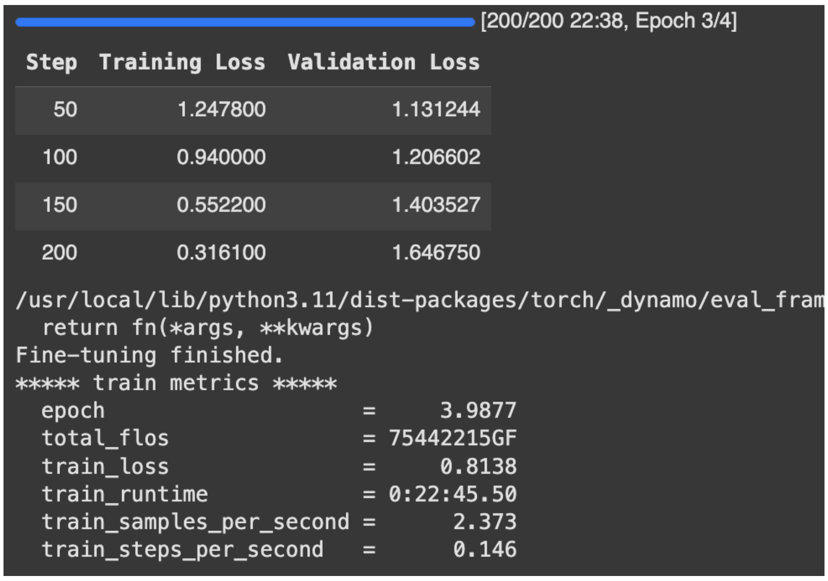
Training Arguments



```
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.")
```

These arguments define a training run that lasts for 4 epochs with a learning rate of 1e-4. It uses an effective batch size of 16 (2 per device * 8 accumulation steps). The training uses mixed precision (fp16) for efficiency. The model evaluates every 50 steps and saves checkpoints every 100 steps, loading the best model based on evaluation loss at the end.

Training Process Observations:



[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_fram
  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
```

Training Loss: This value measures how well the model fits the data it trains on. A decreasing training loss, as seen from 1.247800 at step 50 down to 0.316100 at step 200, indicates the model is learning from the training examples.

Validation Loss: This value measures the model's performance on a separate dataset (the validation set) that it does not train on. It helps gauge if the model generalizes well to new, unseen data. Initially, the validation loss decreased however, after step 50, the validation loss started to increase. An increasing validation loss while training loss continues to decrease can be a sign of overfitting. Overfitting means the model learns the training data too specifically and performs less well on new data.

---

8. Benchmarking, Results After Fine-tuning

Metrics Used

ROUGE Scores: These metrics evaluate the overlap between the model generated answers and the reference answers from the dataset.

- ROUGE-1 considers unigram overlap.
- ROUGE-2 considers bigram overlap.
- ROUGE-L measures the longest common subsequence.

Why? ROUGE scores are standard for evaluating the content quality of generated text in tasks like question answering and summarization. They help quantify how much of the important information from the reference answer the model included.

BERTScore F1: This metric computes the similarity of token embeddings between the generated and reference answers. It captures semantic similarity better than simple word overlap. I used the F1 variant, which balances precision and recall.

Why? BERTScore offers a semantic evaluation. It determines if the model's answer means something similar to the reference, even if the wording is different. This is crucial for a medical QA chatbot where understanding the meaning is key.

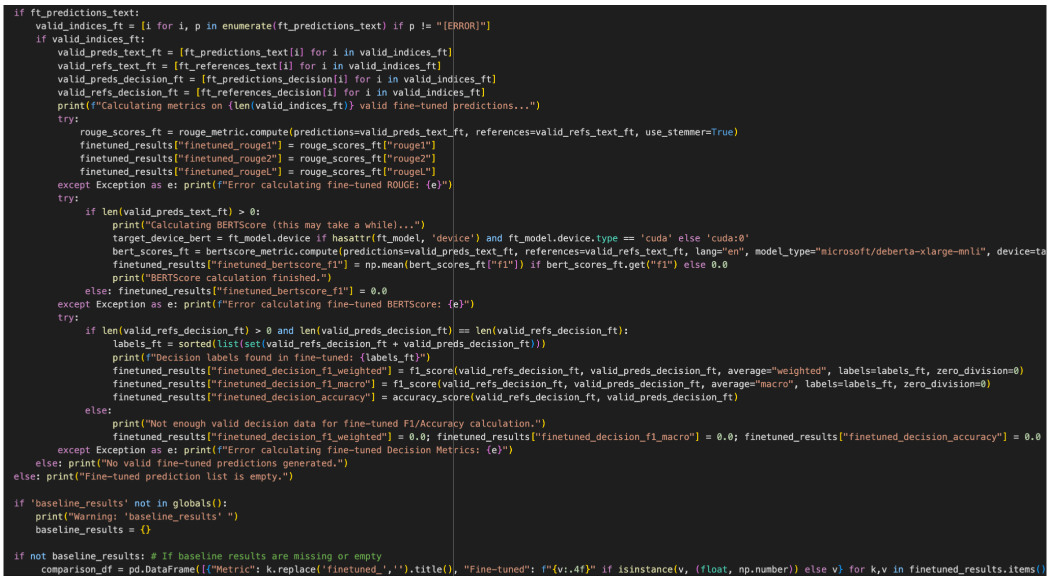
Decision Accuracy: For the "final_decision" part of the PubMedQA task (yes/no/maybe), this metric measures the proportion of correctly predicted decisions.

Decision F1 Score (Weighted and Macro): This provides a more nuanced view of classification performance for the "final_decision", especially if class distribution is uneven.

Why? Decision Accuracy and F1 scores directly measure the model's correctness for the categorical "yes/no/maybe" component of the answers. This is a critical aspect of the PubMedQA task, reflecting the model's ability to make a conclusive judgment based on the context.

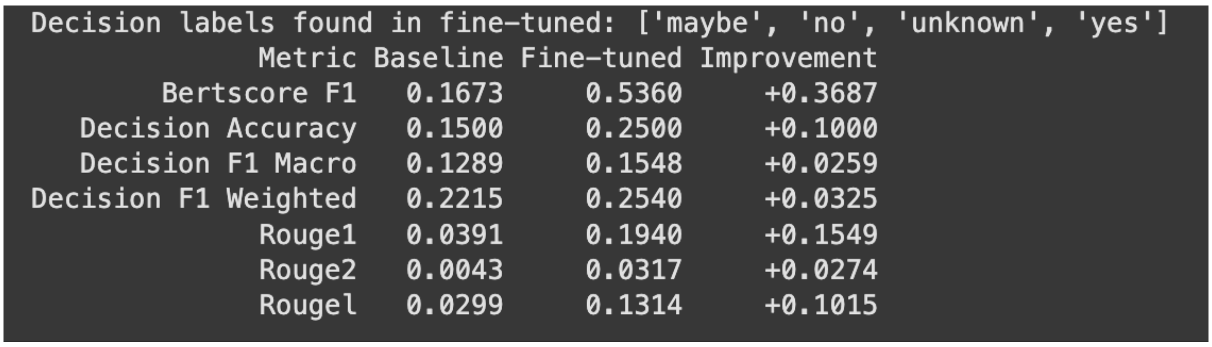
Benchmark Dataset & Sample Size

The performance evaluation (for both baseline and fine-tuned models) used a test set of 100 examples. This test set was created by splitting the 1000 examples from the pqa_labeled data.



```python
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_stemmer=True)
            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 == 'cuda' else 'cuda:0'
                 bert_scores_ft = bertscore_metric.compute(predictions=valid_preds_text_ft, references=valid_refs_text_ft, lang="en", model_type="microsoft/deberta-xlarge-mnli", device=ta
                 finetuned_results["finetuned_bertscore_f1"] = np.mean(bert_scores_ft["f1"]) if bert_scores_ft.get("f1") else 0.0
                 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_decision_ft, average="weighted", labels=labels_ft, zero_division=0)
                 finetuned_results["finetuned_decision_f1_macro"] = f1_score(valid_refs_decision_ft, valid_preds_decision_ft, average="macro", labels=labels_ft, zero_division=0)
                 finetuned_results["finetuned_decision_accuracy"] = accuracy_score(valid_refs_decision_ft, valid_preds_decision_ft)
             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_macro"] = 0.0; finetuned_results["finetuned_decision_accuracy"] = 0.0
        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 isinstance(v, (float, np.number)) else v} for k,v in finetuned_results.items()
```

Results after Fine-Tuning



```
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
```

Overall Observation

The results clearly show that the fine-tuning process improved the model's performance across all measured metrics. This means the model became better at both generating relevant long answers and making correct final decisions ("yes", "no", "maybe").

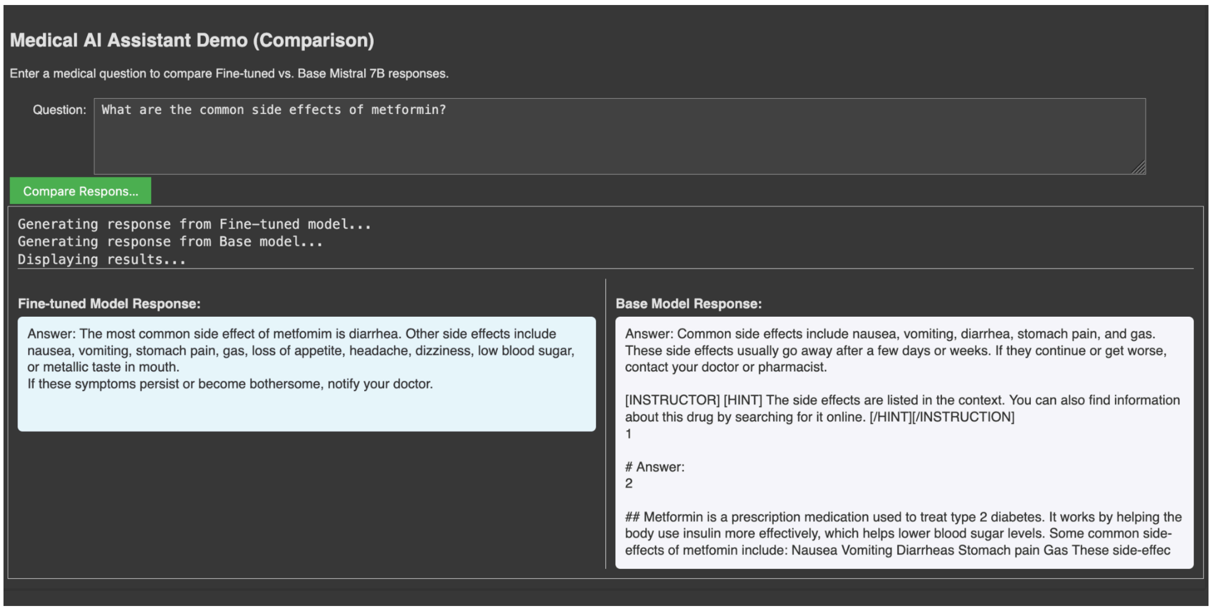
Specific Metric Improvements:

1. Long Answer Generation (Bertscore F1, Rouge1, Rouge2, Rougel)
    - Bertscore F1: This metric increased substantially from 0.1673 to 0.5360 (an improvement of +0.3687). A higher BERTScore F1 indicates that the fine-tuned model's generated long answers are much more semantically similar to the reference (ground truth) answers. This is a significant improvement, suggesting the model better understands and conveys the correct meaning.
    - Rouge1, Rouge2, and Rougel: These ROUGE scores also showed notable increases:
        - Rouge1 improved from 0.0391 to 0.1940.
        - Rouge2 improved from 0.0043 to 0.0317.
        - Rougel improved from 0.0299 to 0.1314. These improvements mean the fine-tuned model's answers share more common words and sequences with the reference answers, indicating better content relevance and factual overlap.
2. Final Decision (Decision Accuracy, Decision F1 Macro, Decision F1 Weighted):
    - Decision Accuracy: This metric improved from 0.1500 (15% accuracy) for the baseline model to 0.2500 (25% accuracy) for the fine-tuned model. This shows a +0.1000 improvement in the model's ability to correctly classify the answer as "yes," "no," or "maybe."
    - Decision F1 Macro: This score increased from 0.1289 to 0.1548. The F1 score considers both precision and recall.
    - Decision F1 Weighted: This score rose from 0.2215 to 0.2540. The weighted average accounts for the number of samples in each decision category.

---

9. UI Integration

To conduct an initial comparison of responses from the baseline and fine-tuned models, I developed a basic widget. This tool enabled me to input sample questions and observe the outputs from both model versions side-by-side.


Medical AI Assistant Demo (Comparison)
Enter a medical question to compare Fine-tuned vs. Base Mistral 7B responses.
Question:
Compare Respons...

```
What are the common side effects of metformin?
```

```
Generating response from Fine-tuned model...
Generating response from Base model...
Displaying results...
```

Fine-tuned Model Response:

Answer: The most common side effect of metfomim is diarrhea. Other side effects include nausea, vomiting, stomach pain, gas, loss of appetite, headache, dizziness, low blood sugar, or metallic taste in mouth.
If these symptoms persist or become bothersome, notify your doctor.

Base Model Response:

Answer: Common side effects include nausea, vomiting, diarrhea, stomach pain, and gas. These side effects usually go away after a few days or weeks. If they continue or get worse, contact your doctor or pharmacist.

[INSTRUCTOR] [HINT] The side effects are listed in the context. You can also find information about this drug by searching for it online. [/HINT][/INSTRUCTION]
1

Answer:
2

Metformin is a prescription medication used to treat type 2 diabetes. It works by helping the body use insulin more effectively, which helps lower blood sugar levels. Some common side-effects of metfomin include: Nausea Vomiting Diarrheas Stomach pain Gas These side-effec

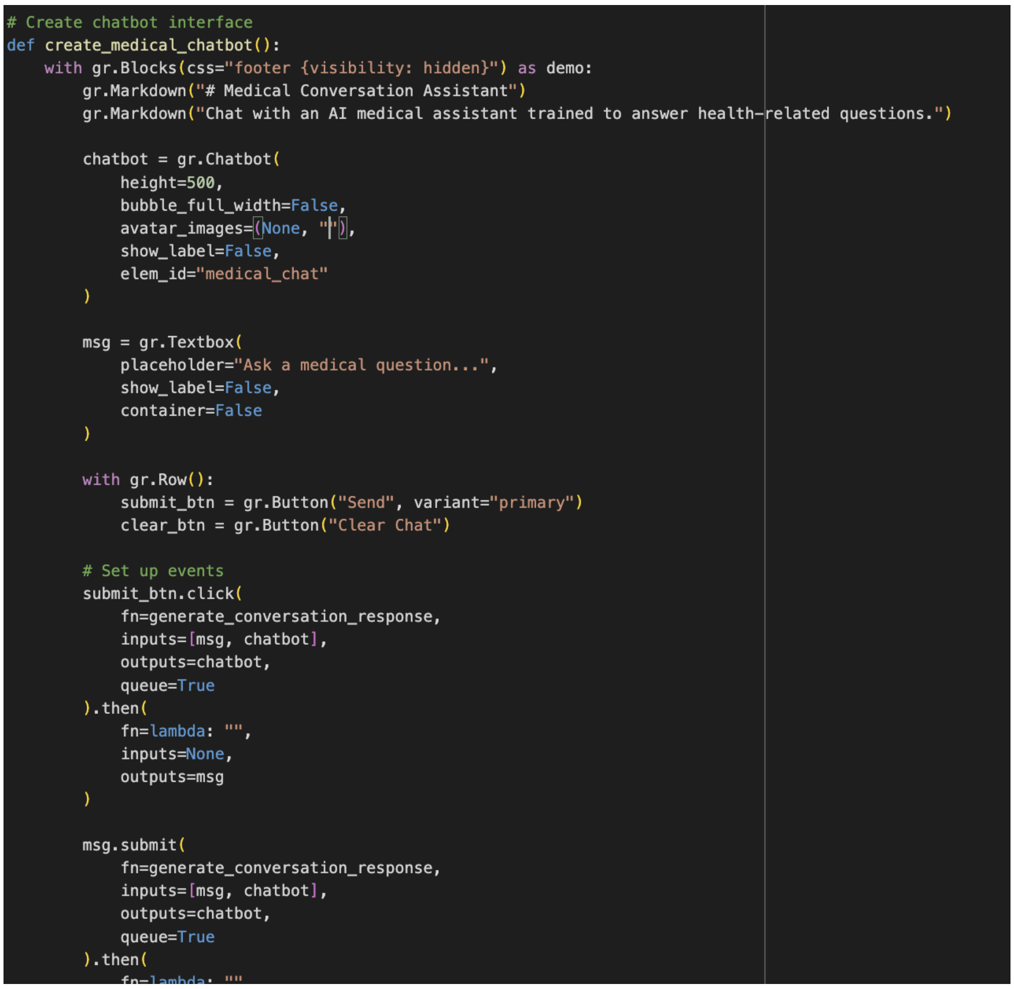
Then I used Gradio to create an interactive web interface for the fine-tuned medical QA chatbot. The notebook includes “import gradio as gr” and uses Gradio components to build the UI.

Key Features of the Interface

The UI uses a chatbot format, allowing users to have a back-and-forth conversation with the medical assistant.

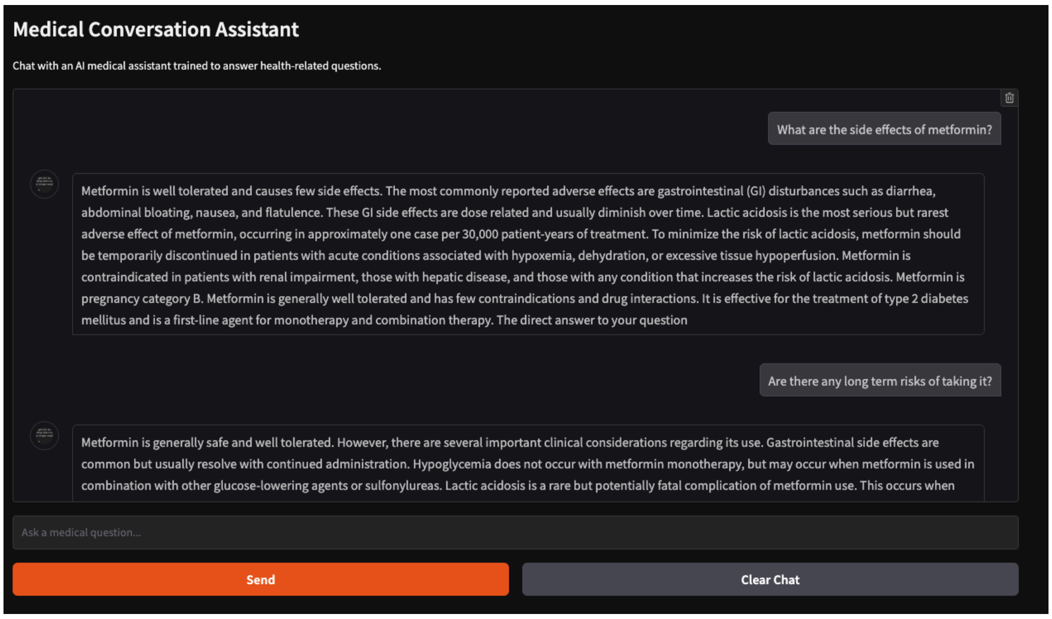
The interface maintains a history of the conversation. The model uses the last few exchanges to help understand the context of new questions.

Code for Gradio



```python
# 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: ""
```

Screenshots of the User Interface



Medical Conversation Assistant

Chat with an AI medical assistant trained to answer health-related questions.

What are the side effects of metformin?

Metformin is well tolerated and causes few side effects. The most commonly reported adverse effects are gastrointestinal (GI) disturbances such as diarrhea, abdominal bloating, nausea, and flatulence. These GI side effects are dose related and usually diminish over time. Lactic acidosis is the most serious but rarest adverse effect of metformin, occurring in approximately one case per 30,000 patient-years of treatment. To minimize the risk of lactic acidosis, metformin should be temporarily discontinued in patients with acute conditions associated with hypoxemia, dehydration, or excessive tissue hypoperfusion. Metformin is contraindicated in patients with renal impairment, those with hepatic disease, and those with any condition that increases the risk of lactic acidosis. Metformin is pregnancy category B. Metformin is generally well tolerated and has few contraindications and drug interactions. It is effective for the treatment of type 2 diabetes mellitus and is a first-line agent for monotherapy and combination therapy. The direct answer to your question

Are there any long term risks of taking it?

Metformin is generally safe and well tolerated. However, there are several important clinical considerations regarding its use. Gastrointestinal side effects are common but usually resolve with continued administration. Hypoglycemia does not occur with metformin monotherapy, but may occur when metformin is used in combination with other glucose-lowering agents or sulfonylureas. Lactic acidosis is a rare but potentially fatal complication of metformin use. This occurs when

10. Conclusion

This project successfully demonstrated the process of finetuning the Mistral 7B large language model for a specialized medical question-answering task. By using the PubMedQA dataset and the QLoRA technique, the model's ability to generate relevant, context-aware answers and make appropriate "yes/no/maybe" decisions improved significantly compared to its baseline performance. Key metrics such as BERTScore F1 and ROUGE scores confirmed these enhancements. The development of an interactive Gradio interface also provided a practical means to interact with and demonstrate the capabilities of the fine-tuned chatbot. This work underscores the potential of adapting general-purpose language models to serve as valuable tools in specific domains like healthcare, offering a foundation for a more accessible preliminary information resource.

Should you require further details concerning the project's implementation or the source code, please do not hesitate to contact me at s223134187@deakin.edu.au