Fine-Tuning Large Language Models for Enterprise Applications

Use Case 4: Sentiment Analysis in Healthcare

---

1. Student Information

- **Student Name:** Mudugamuwa Hewage Bethmi Kisalka
- **Student ID (optional):** s224028677
- **Date Submitted:** 15.05.2025

---

2. Project Introduction

- **Title of the Project:** Fine-Tuning Large Language Models in Enterprise Healthcare Applications – Sentiment Analysis in Healthcare
- **What is the project about?**

  The rapid advancement and widespread adoption of Large Language Models (LLM) have revolutionized enterprise applications across various domains, including customer support, content generation, and data analysis. However, general purpose LLMs often fall short in meeting the nuanced and domain specific demands of industries like healthcare, where accuracy, reliability, and trustworthiness are critical. This project provides with hands-on experience in customizing LLMs through techniques such as Supervised Fine-Tuning (SF) and Low-Rank Adaptation (LoRA), using open source tools like Unsloth.

  Focusing specifically on healthcare, this project explores four key use cases where fine-tuned LLMs can offer impactful improvements: hallucination detection in biomedical question answering systems, detection of medical misinformation in LLM generated content, development of a personalized healthcare question-answering chatbot, and sentiment analysis in healthcare related text.

  I contributed to the Sentiment Analysis in Healthcare use case, which involved analyzing patient feedback and reviews to identify negative sentiments toward medications and treatments.

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

  This project holds significant importance as it addresses the limitations of general purpose LLMs in specialized domains like healthcare. While LLMs have shown great potential in areas such as customer support, content generation, and data analysis, their lack of domain-specific understanding can lead to issues such as hallucinations, misinformation, and generic responses, particularly problematic in high stakes fields like medicine. By fine-tuning these models using techniques like Supervised Fine-Tuning and Low-Rank Adaptation, this project helps bridge the gap between general AI capabilities and the specific needs of enterprise healthcare applications.

  The selected use cases are directly aligned with real-world challenges faced by healthcare providers and digital health platforms. These targeted applications aim to improve model reliability, increase trustworthiness, and ensure safe, accurate communication with patients and professionals. By working with curated datasets and evaluating model performance through

relevant metrics, the project contributes to responsible AI development and helps reduce the risk of incorrect or misleading information in critical decision making contexts.

---

3. API/Token Setup — *Step-by-Step*

Objective: Use of API token for LLM access.

Instructions:

- The model 'flan-t5-large' for Sentiment Analysis in Healthcare does not necessarily require any API/Token setup.
- The Google Colab IDE enables specific access to LLM models without having to create API Tokens.
- Therefore, I did not create any API Tokens to load the model.
- The provider of the 'flan-t5-large' model is:
    - Hugging Face

---

4. Environment Setup

- **Development Platform:**
    - Google Colab
    - Local Machine - Windows
    - GPU Available? [✓] Yes [ ] No
    - GPU Type (if applicable): T4 GPU
- **Python Version: 3.11.12**

```
!python --version
```

```
Python 3.11.12
```

Code: Environment & GPU Check

By using the below command, we can check if GPU is activated or not.

```
import torch
torch.cuda.is_available()
```

```
True
```

---

5. LLM Setup

- **Model Name: Flan-T5 Large**
- **Provider (OpenAI, Hugging Face, etc.): Hugging Face**
- **Key Libraries & Dependencies (with versions):**

- transformers : 4.51.3
- torch: 2.6.0+cu124
- datasets: 2.14.4
- accelerate: 1.6.0
- sentencepiece: 0.2.0
- peft: 0.15.2
- gradio: 5.29.0

- **Libraries and Dependencies Required:**
  The below link provides the 'requirements.txt' which contains all the key libraries and dependencies required with their necessary versions.

  https://drive.google.com/file/d/1Laq73iWN8V4VJv-QV6zDFRM5NVKiVxFo/view?usp=sharing

Code: Install & Import

Using the below commands, we can install the necessary libraries.

```
# install libraries
!pip install transformers[torch] datasets accelerate torch sentencepiece peft
```

```
!pip install gradio
```

---

6. Dataset Description

- **Dataset Name & Source:** Drug Reviews (Druglib.com) by UCI Machine Learning Repository
- **Access Link (if public):**
  https://archive.ics.uci.edu/dataset/461/drug+review+dataset+druglib+com
- **Feature Dictionary / Variable Description:**
  - The Drug Review dataset consists of 4143 observations
  - Consists of total of 8 variables:
    - reviewID: Review ID
    - urlDrugName: Drug Name
    - Rating: Rating from 1 to 10
    - Effectiveness: Effectiveness of the Drug
    - sideEffects: Side effects of the Drug
    - condition: What the drug is used for
    - benefitsReview: Review of benefits
    - sideEffectsReview: Description of side effects
    - commentsReview: General review of the drug
    - sentiment: Sentiment label of the drug

- The variables of interest for the analysis and fine-tuning purposes are ,
  - Rating
  - commentsReview

- **Was preprocessing done? If yes, describe:**
  - The ratings were converted to labels Positive, Negative, and Neutral as below:
    - Ratings from 7-10: Positive
    - Ratings from 5-6: Neutral
    - Ratings: 1-4: Negative
  - Removed missing values.

Code: Load & Preprocess Dataset

```
# labeling rates to positive, neutral and negative
def classify_label(rating):
  if rating>=7:
    return "positive"
  elif rating>=5:
    return "neutral"
  else:
    return "negative"
```

---

7. Improving LLM Performance

The fine-tuning process of the Flan-T5-Large model was done in five stages as below.

Stage	LoRA Configuration	Impute Class Imbalance	No.of Epochs	Results (Accuracy)
Stage 1 (Initial model with no training)	No LoRA	No	No	55%
Stage 2 (First Fine-tuning)	lora_r= 16 lora_alpha=32 lora_dropout=0.1 learning_rate=1e-4 batch_size: = 4 epochs: = 5	No	5	52%
Stage 3	lora_r= 16 lora_alpha=32 lora_dropout=0.1 learning_rate=1e-4 batch_size: = 4 epochs: = 5	Yes	5	57%

Stage 4	lora_r= 32 lora_alpha=64 lora_dropout=0.1 learning_rate=5e-5 batch_size: = 4 epochs: = 8	Yes	8	57% (Better Precision and recall values across classes)
Stage 5 (Use of prompt when tokenizing data)	lora_r= 32 lora_alpha=64 lora_dropout=0.1 learning_rate=5e-5 batch_size: = 4 epochs: = 8	Yes	8	60% (High Precision and recall on negative reviews)

Code Snippets for Each Step

1. Load the model

```
#load model
model_name = 'google/flan-t5-large'
tokenizer = T5Tokenizer.from_pretrained(model_name)
model = T5ForConditionalGeneration.from_pretrained(model_name, device_map="auto")
```

2. Stage 1

```
# predict sentiment
def predict_sentiment(text):
    #input_text = f"Classify Sentiment: {text}"
    input_text = f"Classify the sentiment of the following text as Positive, Negative, or Neutral: {text}. Output only one of these labels."

    inputs = tokenizer(input_text, return_tensors="pt", padding=True, truncation=True, max_length=512)

    inputs = {key: value.to(device) for key, value in inputs.items()}

    with torch.no_grad():
        outputs = model.generate(**inputs)

    sentiment = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return sentiment
```

3. Stage 2

- Config.py: Save model configurations including LoRA configurations, training configurations.

```python
##config.py
%%writefile config.py
from dataclasses import dataclass

@dataclass
class ModelConfig:
    # model configurations
    model_name: str = "google/flan-t5-large"
    num_labels: int = 3  # positive, negative, neutral

    # training configurations
    learning_rate: float = 1e-4
    batch_size: int = 4  # small batch size for Colab
    epochs: int = 5

    # LoRA configurations
    lora_r: int = 16
    lora_alpha: int = 32
    lora_dropout: float = 0.1

    # paths
    train_data_path: str = "data/new_train.csv"
    test_data_path: str = "data/new_test.csv"

    # Output
    output_dir: str = "sentiment_model"
```

- data_loader.py: Tokenize data and save tokenized data as cached data. This prevents from tokenizing the data every time the model is fine-tuned.

```python
%%writefile src/data_loader.py

import os
import pandas as pd
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import AutoTokenizer
from config import ModelConfig
import pickle

class SentimentDataset(Dataset):
    def __init__(self, input_ids, attention_masks, labels):
        self.input_ids = input_ids
        self.attention_masks = attention_masks
        self.labels = labels

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        return {
            'input_ids': self.input_ids[idx],
            'attention_mask': self.attention_masks[idx],
            'labels': self.labels[idx]
        }
```

```python
def tokenize_and_cache_data(datapath, tokenizer, max_length=128, cache_dir='/content/drive/My Drive/Deakin_units/T1-2025/SIT764/sentiment-analysis/cached_data'):
    """
    Tokenize data and save to cache for faster loading

    Args:
    - datapath (str): Path to CSV file
    - tokenizer (AutoTokenizer): Tokenizer to use
    - max_length (int): Maximum sequence length
    - cache_dir (str): Directory to save cached files

    Returns:
    - Tuple of (input_ids, attention_masks, labels)
    """
    # Create cache directory if it doesn't exist
    os.makedirs(cache_dir, exist_ok=True)

    # Generate cache filename
    cache_filename = os.path.basename(datapath).replace('.csv', '_tokenized.pkl')
    cache_path = os.path.join(cache_dir, cache_filename)

    # Check if cached file exists
    if os.path.exists(cache_path):
        print(f"Loading cached tokenized data from {cache_path}")
        with open(cache_path, 'rb') as f:
            return pickle.load(f)

    # Read CSV
    df = pd.read_csv(datapath)

    # Prepare texts and labels
    texts = df['commentsReview'].tolist()
```

```
    # Map labels to integers
    label_map = {'negative': 0, 'neutral': 1, 'positive': 2}
    labels = [label_map[label.lower()] for label in df['sentiment']]

    # Tokenize texts
    encodings = tokenizer(
        texts,
        return_tensors='pt',
        max_length=max_length,
        padding='max_length',
        truncation=True
    )

    # Extract input_ids and attention_masks
    input_ids = encodings['input_ids']
    attention_masks = encodings['attention_mask']

    # Convert to tensor
    labels = torch.tensor(labels, dtype=torch.long)

    # Cache the tokenized data
    cached_data = (input_ids, attention_masks, labels)
    with open(cache_path, 'wb') as f:
        pickle.dump(cached_data, f)

    print(f"Tokenized data cached to {cache_path}")
    return cached_data
```

```
def create_dataloaders(config, tokenizer):
    # Tokenize and cache train data
    train_input_ids, train_attention_masks, train_labels = tokenize_and_cache_data(
        config.train_data_path,
        tokenizer
    )

    # Tokenize and cache test data
    test_input_ids, test_attention_masks, test_labels = tokenize_and_cache_data(
        config.test_data_path,
        tokenizer
    )

    # Create datasets
    train_dataset = SentimentDataset(
        train_input_ids,
        train_attention_masks,
        train_labels
    )
    test_dataset = SentimentDataset(
        test_input_ids,
        test_attention_masks,
        test_labels
    )

    # Create dataloaders
    train_loader = DataLoader(
        train_dataset,
        batch_size=config.batch_size,
        shuffle=True
```

```
    )
    test_loader = DataLoader(
        test_dataset,
        batch_size=config.batch_size,
        shuffle=False
    )

    return train_loader, test_loader
```

- Model.py: Model configurations. Use Sequence to Classification as task type in LoRA.

```
# model configuration

%%writefile src/model.py

import torch
from transformers import AutoModelForSequenceClassification
from peft import LoraConfig, get_peft_model, TaskType

def create_model(config):
    # Load pre-trained model
    model = AutoModelForSequenceClassification.from_pretrained(
        config.model_name,
        num_labels=config.num_labels
    )

    # LoRA configuration
    lora_config = LoraConfig(
        task_type=TaskType.SEQ_CLS,
        r=config.lora_r,
        lora_alpha=config.lora_alpha,
        lora_dropout=config.lora_dropout,
        target_modules=['q', 'v']  # Typically query and value projection layers
    )

    # Apply LoRA
    peft_model = get_peft_model(model, lora_config)

    return peft_model
```

```
def count_trainable_parameters(model):
    trainable_params = sum(
        p.numel() for p in model.parameters() if p.requires_grad
    )
    total_params = sum(p.numel() for p in model.parameters())

    print(f"Trainable parameters: {trainable_params}")
    print(f"Total parameters: {total_params}")
    print(f"Percentage of trainable parameters: {trainable_params/total_params*100:.2f}%")
```

- Train.py: Train and validate and calculate loss at each epoch and calculate evaluation metrics.

```
%%writefile src/train.py

import torch
from torch.optim import AdamW
from transformers import get_linear_schedule_with_warmup
from tqdm import tqdm

def train_epoch(model, train_loader, optimizer, scheduler, device):
    model.train()
    total_loss = 0

    for batch in tqdm(train_loader, desc="Training"):
        # move batch to device
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        # zero gradients
        optimizer.zero_grad()

        # forward pass
        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            labels=labels
        )
        loss = outputs.loss
        total_loss += loss.item()
```

```
        # backward pass
        loss.backward()
        optimizer.step()
        scheduler.step()

    return total_loss / len(train_loader)

def train_model(model, train_loader, test_loader, config, device):
    # Prepare optimizer and schedule
    optimizer = AdamW(
        model.parameters(),
        lr=config.learning_rate
    )

    # Total training steps
    total_steps = len(train_loader) * config.epochs
    scheduler = get_linear_schedule_with_warmup(
        optimizer,
        num_warmup_steps=total_steps // 10,
        num_training_steps=total_steps
    )

    best_accuracy = 0

    for epoch in range(config.epochs):
        print(f"Epoch {epoch+1}/{config.epochs}")

        # Training
        train_loss = train_epoch(
            model, train_loader, optimizer, scheduler, device
        )
```

```
        # Evaluation
        val_results = evaluate_model(
            model, test_loader, device
        )

        print(f"Train Loss: {train_loss:.4f}")
        print(f"Validation Accuracy: {val_results['accuracy']:.4f}")

        # Save best model
        if val_results['accuracy'] > best_accuracy:
            best_accuracy = val_results['accuracy']
            model.save_pretrained(config.output_dir)
            print(f"New best model saved with accuracy: {best_accuracy:.4f}")

    return model

def evaluate_model(model, test_loader, device):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Evaluating"):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            outputs = model(
                input_ids=input_ids,
                attention_mask=attention_mask
            )
```

```
            preds = torch.argmax(outputs.logits, dim=1)

            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    # Import evaluation metrics here to avoid circular imports
    from sklearn.metrics import (
        accuracy_score,
        precision_score,
        recall_score,
        f1_score,
        confusion_matrix
    )

    results = {
        'accuracy': accuracy_score(all_labels, all_preds),
        'precision': precision_score(all_labels, all_preds, average='weighted'),
        'recall': recall_score(all_labels, all_preds, average='weighted'),
        'f1_score': f1_score(all_labels, all_preds, average='weighted'),
        'confusion_matrix': confusion_matrix(all_labels, all_preds)
    }

    return results
```

- Main.py

```
def main():
    # Configuration
    config = ModelConfig()

    # Device
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")

    # Tokenizer
    tokenizer = AutoTokenizer.from_pretrained(config.model_name)

    # Create Dataloaders
    train_loader, test_loader = create_dataloaders(config, tokenizer)

    # Create Model
    model = create_model(config)
    model.to(device)

    # Count Parameters
    count_trainable_parameters(model)

    # Train Model
    trained_model = train_model(
        model, train_loader, test_loader, config, device
    )
```

```
# Final Evaluation
results = evaluate_model(trained_model, test_loader, device)

# Print Results
print("\n--- Final Evaluation Results ---")
for key, value in results.items():
    if key != 'confusion_matrix':
        print(f"{key}: {value}")

# Plot Confusion Matrix
plt.figure(figsize=(10,7))
sns.heatmap(
    results['confusion_matrix'],
    annot=True,
    fmt='d',
    cmap='Blues'
)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.savefig(f"{config.output_dir}/confusion_matrix.png")
```

```
if __name__ == "__main__":
    main()
```

4. Stage 3: Impute Class imbalance into model weights and use of Cross Entropy Loss function.

```
import torch
import os
import seaborn as sns
import matplotlib.pyplot as plt
from datetime import datetime
import numpy as np
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import classification_report
from config import ModelConfig
from transformers import AutoTokenizer
from src.data_loader import create_dataloaders
from src.model import create_model, count_trainable_parameters
from src.evaluate import evaluate_model

def train_model(model, train_loader, test_loader, config, device,tokenizer):
    all_labels = torch.cat([batch['labels'] for batch in train_loader])
    class_weights = compute_class_weight('balanced', classes=np.unique(all_labels.cpu().numpy()), y=all_labels.cpu().numpy())
    weights = torch.tensor(class_weights, dtype=torch.float).to(device)
    loss_fn = torch.nn.CrossEntropyLoss(weight=weights)
    optimizer = torch.optim.AdamW(model.parameters(), lr=config.learning_rate)
```

5. Stage 4: Increase number of epochs and changed LoRA configurations

```
# config.py
%%writefile config.py
from dataclasses import dataclass

@dataclass
class ModelConfig:
    # model configurations
    model_name: str = "google/flan-t5-large"
    num_labels: int = 3  # positive, negative, neutral

    # training configurations
    learning_rate: float = 5e-5
    batch_size: int = 4  # small batch size for Colab
    epochs: int = 8

    # LoRA configurations
    lora_r: int = 32
    lora_alpha: int = 64
    lora_dropout: float = 0.1

    # paths
    train_data_path: str = "data/new_train.csv"
    test_data_path: str = "data/new_test.csv"

    # Output
    output_dir: str = "sentiment_model"
```

6. Stage 5: Addition of a prompt when tokenizing data.

```
df = pd.read_csv(datapath)
prompt_prefix = "Classify the sentiment as positive, negative, or neutral: "
texts = [prompt_prefix + str(text) for text in df['commentsReview'].tolist()]
# texts = df['commentsReview'].tolist()
label_map = {'negative': 0, 'neutral': 1, 'positive': 2}
labels = [label_map[label.lower()] for label in df['sentiment']]
```

8. Benchmarking & Evaluation

Required Components:

- **Metrics Used (e.g., Accuracy, F1, BLEU, etc.):** Accuracy, F1-Score, Recall, Precision, Validation and Training Loss
- **Why those metrics?**
    - For sentiment analysis using a fine-tune LLM model, metrics like accuracy, confusion matrix, precision, recall, F1-Score, and training/validation loss are well suited because they provide a comprehensive view of model performance. Accuracy offers a general measure of correctness, while the confusion matrix breaks down performance across classes to highlight specific misclassifications. Precision and recall are critical when class imbalance exists, helping assess the cost of false positives and false negatives, respectively. The F1-score balances these two and is especially useful when both types of errors matter. Monitoring training and validation loss helps detect overfitting or underfitting, ensuring the model generalizes well to unseen data. Together these metrics support informed model selection and evaluation.
- **Benchmark Dataset & Sample Size:**
    - Test Data: Drug Reviews (Druglib.com) by UCI Machine Learning Repository.
    - Sample Size: 1036 Records

Code: Metric Calculation

```
# Import evaluation metrics here to avoid circular imports
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix
)

results = {
    'accuracy': accuracy_score(all_labels, all_preds),
    'precision': precision_score(all_labels, all_preds, average='weighted'),
    'recall': recall_score(all_labels, all_preds, average='weighted'),
    'f1_score': f1_score(all_labels, all_preds, average='weighted'),
    'confusion_matrix': confusion_matrix(all_labels, all_preds)
}
```

```
# Print Results
print("\n--- Final Evaluation Results ---")
for key, value in results.items():
    if key != 'confusion_matrix':
        print(f"{key}: {value}")

# Plot Confusion Matrix
plt.figure(figsize=(10,7))
sns.heatmap(
    results['confusion_matrix'],
    annot=True,
    fmt='d',
    cmap='Blues'
)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.savefig(f"{config.output_dir}/confusion_matrix.png")
```

```
train_losses = []
val_losses = []

for epoch in range(config.epochs):
    model.train()
    total_loss = 0

    for batch in train_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        optimizer.zero_grad()
        outputs = model(input_ids=input_ids, attention_mask=attention_mask)
        loss = loss_fn(outputs.logits, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    avg_train_loss = total_loss / len(train_loader)
    train_losses.append(avg_train_loss)
    print(f"Epoch {epoch+1}/{config.epochs}, Training Loss: {avg_train_loss:.4f}")

    results = evaluate_model(model, test_loader, device)
    val_loss = results['loss']
    val_losses.append(val_loss)
```

Plot Results

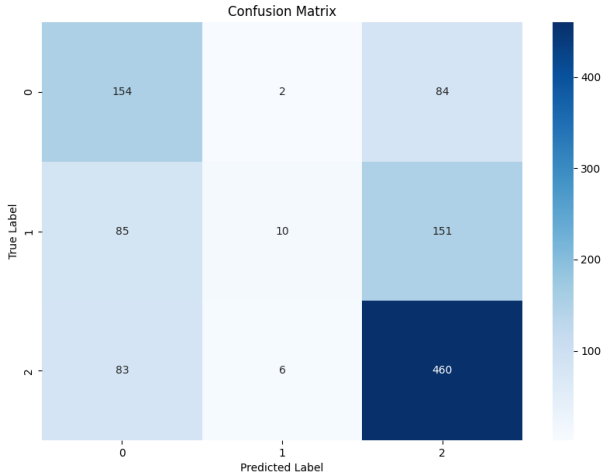
Plot Confusion Metrix:

```
# Confusion matrix
plt.figure(figsize=(10,7))
sns.heatmap(results['confusion_matrix'], annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
os.makedirs(config.output_dir, exist_ok=True)
plt.savefig(os.path.join(config.output_dir, 'confusion_matrix.png'))
```

Plot Validation and Training Loss:

```
# Plot training vs validation loss
plt.figure(figsize=(10, 6))
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training vs Validation Loss')
plt.legend()
plt.grid(True)
plt.savefig(os.path.join(session_dir, 'loss_plot.png'))
plt.close()
```

Interpretation:



The confusion matrix provides a detailed breakdown of the model's predictions versus actual labels. It tells how many predictions were correct for each class and what types of errors the model did.

Training and Validation Loss:

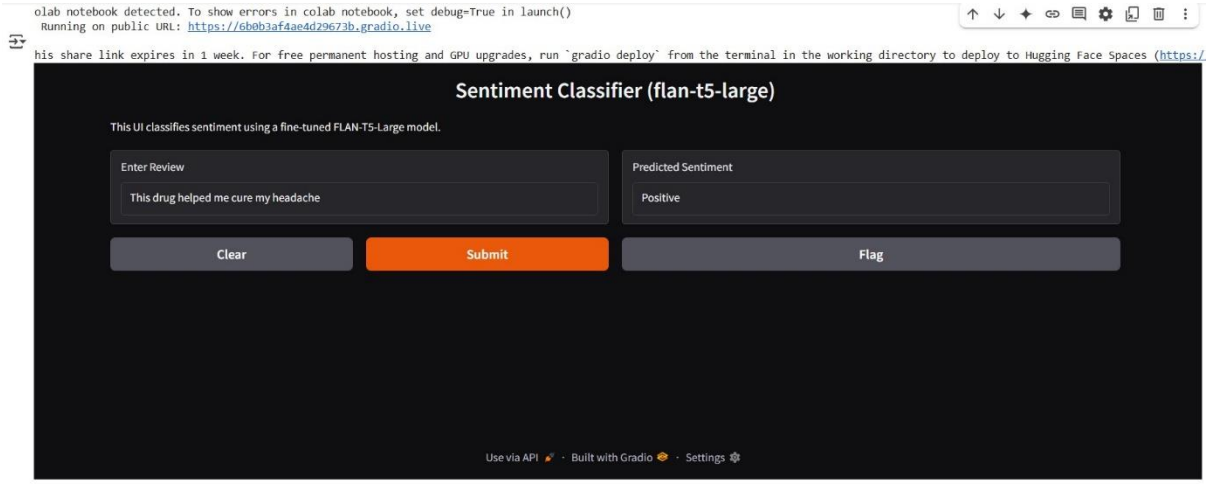
The Training and Validation loss plot shows how well the model is learning over each epoch. The training loss measures the model's error on the training set, while the validation loss measures the model's error on the validation (unseen) set. If both losses are decreasing the model is learning well.

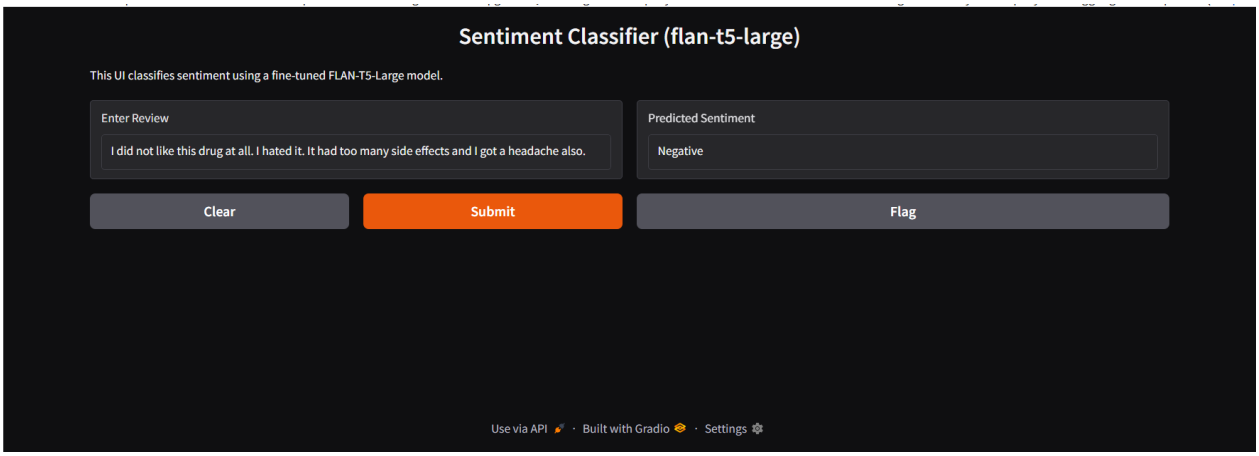
---

9. UI Integration

- **Tool Used (e.g., Gradio, Streamlit):** Gradio
- **Key Features of the Interface:**
    - Ability to enter any type of drug review comment.
    - Predicts the sentiment of any desired drug review.
    - User friendly.

- **Screenshots of Working UI:**





Code: UI Implementation:

- **Load base model and tokenizer with fine-tuned LoRA adapter.**

```
from transformers import T5Tokenizer, T5ForConditionalGeneration
from peft import PeftModel, PeftConfig
import torch

# Step 1: Paths
adapter_path = "/content/drive/My Drive/Deakin_units/T1-2025/SIT764/sentiment-analysis/flan-t5-large/sentiment_model/session_20250509_021444/"
base_model_name = "google/flan-t5-large"  # same as used during training

# Step 2: Load base model and tokenizer
tokenizer = T5Tokenizer.from_pretrained(base_model_name)
base_model = T5ForConditionalGeneration.from_pretrained(base_model_name)

# Step 3: Load the adapter
model = PeftModel.from_pretrained(base_model, adapter_path)
model.eval()

# Step 4: Device setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

- **Import gradio and create the UI**

```
valid_labels = ["negative", "neutral", "positive"]

def predict_sentiment(text):
    prompt = f"Classify the sentiment of the following text as positive, negative, or neutral: {text}. Output only one of these labels."
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True, padding=True).to(device)

    with torch.no_grad():
        output_ids = model.generate(**inputs, max_new_tokens=5)
        output_text = tokenizer.decode(output_ids[0], skip_special_tokens=True).strip().lower()

        # Normalize to valid label
        for label in valid_labels:
            if label in output_text:
                return label.capitalize()

        return "Unknown"
```

```
iface = gr.Interface(
    fn=predict_sentiment,
    inputs=gr.Textbox(label="Enter Review"),
    outputs=gr.Textbox(label="Predicted Sentiment"),
    title="Sentiment Classifier (flan-t5-large)",
    description="This UI classifies sentiment using a fine-tuned FLAN-T5-Large model."
)
```

```
iface.launch()
```