**Laguna State Polytechnic University Sta Cruz Campus**

**College of Computer Studies**

**CALORIE WISE SUPPORT**

**Chatbot**

Submitted by:

**Brent Justine Barbadillo**

**Ernesto Candelaria**

**Azzaleah Faith Oxina**

Submitted to:

**Mr. Mark Bernardino**

**Mrs. Mia Villarica**

**Mr. Romel Seranno**

Video Documentation: **CalorieWise Chatbot Project Process**

The CalorieWise Chatbot operates within the domain of nutrition and dietary management, aiming to assist users in making informed dietary choices by providing real-time calorie information and meal suggestions. The reasoning method employed in this project integrates a hybrid approach that combines rule-based logic for straightforward queries with advanced machine learning techniques, specifically utilizing the Llama-3.2 transformer model from the unsloth library. This model is particularly effective for natural language processing tasks, allowing the chatbot to understand and generate human-like responses based on user inputs, thereby enhancing user interaction and engagement in dietary conversations.

**Data Preparation**

Let's dive into the data preparation phase.

The dataset used for training the CalorieWise Chatbot is called the \*\*DietAI Dataset\*\*, which contains various conversations about dietary preferences and calorie information.

During preprocessing, I performed several steps:

- Tokenization: This involves breaking down the text into smaller units (tokens) that can be processed by the model.

- Normalization: This step ensures that the text is consistent in format, such as converting all text to lowercase.

Here’s a snippet of the code used for preprocessing:

**```python**

**from unsloth.chat\_templates import get\_chat\_template**

**tokenizer = get\_chat\_template(tokenizer, chat\_template="llama-3.1")**

**def formatting\_prompts\_func(examples):**

**convos = examples["conversations"]**

**texts = [tokenizer.apply\_chat\_template(convo, tokenize=False, add\_generation\_prompt=False) for convo in convos]**

**return {"text": texts}**

**from datasets import load\_dataset**

**dataset = load\_dataset("sfardin/dietAi-dataset-expanded", split="train")**

**dataset = dataset.map(formatting\_prompts\_func, batched=True)**

**```**

**Explanation of Code:**

- The `get\_chat\_template` function initializes a chat template for tokenization.

- The `formatting\_prompts\_func` processes each conversation, applying the chat template to structure the data appropriately.

- The dataset is loaded using `load\_dataset`, specifying the source dataset and split. Finally, we map our formatting function over the dataset to prepare it for training.

**Model Implementation (3-4 minutes)**

Now let's discuss the model implementation.

I utilized a \*transformer-based architecture\* with a focus on fine-tuning capabilities. The selected reasoning technique combines rule-based responses for specific queries with a neural network to handle more complex interactions.

Here’s a key section of the code that sets up the model:

**```python**

**from unsloth import FastLanguageModel**

**model, tokenizer = FastLanguageModel.from\_pretrained(**

**model\_name="unsloth/Llama-3.2-1B-Instruct-bnb-4bit",**

**max\_seq\_length=2048,**

**load\_in\_4bit=True**

**)**

**```**

**Explanation of Code:**

- This code imports `FastLanguageModel` from `unsloth` and initializes it with a pre-trained model.

- The `max\_seq\_length` parameter controls how long input sequences can be, while `load\_in\_4bit` optimizes memory usage by loading weights in 4-bit precision.

**Training the Model**

Next, let’s move on to training the model.

The training process involved setting various parameters such as:

- Epochs: The number of times the learning algorithm will work through the entire training dataset.

- Learning Rate: A hyperparameter that controls how much to change the model in response to estimated errors each time weights are updated.

Here’s a snippet showcasing how I set up training:

**```python**

**from transformers import TrainingArguments**

**from trl import SFTTrainer**

**trainer = SFTTrainer(**

**model=model,**

**tokenizer=tokenizer,**

**train\_dataset=dataset,**

**args=TrainingArguments(**

**per\_device\_train\_batch\_size=2,**

**num\_train\_epochs=1,**

**learning\_rate=2e-4,**

**output\_dir="outputs"**

**)**

**)**

**trainer.train()**

**```**

**Explanation of Code:**

- `SFTTrainer` is initialized with our model, tokenizer, and training dataset.

- The `TrainingArguments` class specifies parameters like batch size, number of epochs, learning rate, and output directory.

- Finally, we call `trainer.train()` to begin training our model using these settings.

**Testing and Evaluation**

Next, let’s see how we tested and evaluated our chatbot.

I implemented several test cases to evaluate its performance in real-time conversations. Key performance metrics included:

- Accuracy: The overall correctness of predictions.

- Precision: The ratio of correctly predicted positive observations to total predicted positives.

- Recall: The ratio of correctly predicted positive observations to all actual positives.

Here’s an example of how evaluation metrics were calculated:

**```python**

**from sklearn.metrics import precision\_score, recall\_score, accuracy\_score**

**def evaluate\_metrics(true\_labels, predicted\_labels):**

**precision = precision\_score(true\_labels, predicted\_labels, average='weighted')**

**recall = recall\_score(true\_labels, predicted\_labels, average='weighted')**

**accuracy = accuracy\_score(true\_labels, predicted\_labels)**

**return {'precision': precision, 'recall': recall, 'accuracy': accuracy}**

**```**

**Explanation of Code:**

- This function calculates precision, recall, and accuracy based on true and predicted labels using functions from `sklearn.metrics`.

- It returns a dictionary containing these metrics for easy reference.

**Challenges**

Throughout this project, I encountered several challenges. One significant issue was managing GPU memory during training due to high computational demands. To overcome this, I implemented gradient checkpointing and optimized batch sizes.

Additionally, fine-tuning hyperparameters required multiple iterations before achieving satisfactory performance.

**Conclusion**

To wrap up, this project successfully developed a functional CalorieWise Chatbot capable of assisting users with dietary inquiries.

In future iterations, I would suggest exploring larger datasets for improved accuracy and incorporating user feedback mechanisms to enhance response quality. Thank you for watching my documentation on this exciting project!

Citations:

<https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/30453813/f709ad42-a94e-4746-82a0-908e34a91f74/caloriewisechatbot.py>