High Level Design

Product Chat Assistant

Written By	Om Singh, Shaileja Patil
Document Version	1.5
Last Revised Date	13 – June -2024

Document Control

Change Record:

Version	Date	Author	Comment
1.0	26 April 2024	Om Singh	Initial project setup and data preparation
1.1	10 May 2024	Shaieja Patil	Data preprocessing completed and dataset finalized
1.2	25 May 2024	Om Singh	Fine-tuning of LLaMA 2 model using UnSloth
1.3	9 June 2024	Shaieja Patil	Development of Streamlit application
1.4	27 June 2024	Om Singh	Application testing and integration with the model
1.5	13 July 2024	Shaieja Patil	Final project documentation and preparation for submission

Contents

1. Introduction

The Product Chat Assistant is an innovative AI-driven application designed to revolutionize the online shopping experience. It serves as an intelligent virtual assistant capable of interacting with users in a natural and engaging manner. By leveraging advanced natural language processing (NLP) technologies, the assistant can provide detailed product information, answer user queries, and even negotiate prices, thereby enhancing customer satisfaction and driving sales.

Key Features :

1. Product Information Retrieval:

The assistant can retrieve detailed information about various products, including specifications, features, pricing, and availability. This allows users to make informed decisions quickly and easily.

2. Interactive Conversations:

The chat assistant engages users in natural, human-like conversations. It understands and responds to queries contextually, making interactions seamless and intuitive.

3. Price Negotiation:

A unique feature of this assistant is its ability to negotiate prices. Users can haggle over product prices or terms, similar to a real-world shopping experience.

4. Personalized Recommendations:

By analyzing user preferences and browsing history, the assistant can offer personalized product recommendations, enhancing the shopping experience.

5. Multi-Channel Support:

The assistant can be integrated into various platforms, including websites, mobile apps, and social media channels, ensuring consistent user experience across different touchpoints.

2. Scope

The Product Chat Assistant project aims to develop a conversational AI assistant that provides personalized product information and support to customers. The scope of this project includes:

Functional Requirements

* Develop a chat interface for customers to ask questions about products
* Integrate a fine-tuned LLaMA 2 model to generate responses to customer Queries
* Implement a bargaining feature to negotiate prices or other terms
* Provide product information, including descriptions, prices, and reviews

Data Requirements

* Collect and preprocess a dataset of 370 rows and 12 columns using OpenAI and Gemini
* Append conversation script and input prompt features to the dataset
* Push the dataset to Hugging Face for model training

Technical Requirements

* Use UnSloth's FastLanguageModel for fine-tuning LLaMA 2
* Implement SFTTrainer for training the model
* Deploy the trained model using Streamlit for frontend development

Deliverables

* A functional Product Chat Assistant with a user-friendly interface
* A fine-tuned LLaMA 2 model trained on the collected dataset
* A dataset pushed to Hugging Face for future use

Timeline

* Data collection and preprocessing: 1 weeks
* Model training and fine-tuning: 3 weeks
* Frontend development and deployment: 1 weeks

Resources

* Project lead: Mandalor09
* Technical resources: UnSloth, Hugging Face, Streamlit
* Data resources: OpenAI, Gemini

3. Architecture

The architecture used can be broken down into the following components:

1. Data Generation:

This component is responsible for generating data using OpenAI and Gemini.
This data is likely to be a large dataset of text samples that will be used to train the conversational AI model.

2. Data Preprocessing:

After generating the data, it needs to be preprocessed to make it suitable for training the model. This involves cleaning the data, adding relevant features, and transforming it into a format that can be used by the model.

3. Model Training:

The preprocessed data is then used to fine-tune a pre-trained LLAMA model using UnSloth.
The fine-tuning process involves adjusting the model's parameters to fit the specific task of conversational AI.

4. Conversation Script and Input Prompt:

The fine-tuned model is then used to generate a conversation script and input prompt.
This script and prompt are likely to be used to interact with the model and generate responses.

5. Dataset Upload:

The preprocessed dataset is uploaded to Hugging Face Hub, a platform for hosting and sharing machine learning models.

6. Model Deployment:

The fine-tuned LLAMA model is deployed on Hugging Face Hub, making it accessible for use in applications.

7. User Interface:

A user interface is built using Streamlit, a Python library for building web applications. This interface allows users to interact with the deployed model, providing input prompts and receiving responses.

The architecture can be summarized as follows:

Data Generation -> Data Preprocessing -> Model Training -> Conversation Script and Input Prompt -> Dataset Upload -> Model Deployment -> User Interface

This architecture is a typical pipeline for building and deploying a conversational AI model, and it involves data generation, preprocessing, model training, deployment, and user interface development.

3.1) Data Preparation Using OpenAI and Gemini

Data Sources

To create a comprehensive dataset, we will use data from two main sources:
- OpenAI
- Gemini

Data Extraction

i) OpenAI

We will utilize OpenAI's capabilities for text generation and API calls to extract relevant product data. OpenAI's API will be called to generate product information such as descriptions, reviews, and ratings.

Ii) Gemini

Similarly, Gemini's API will be used to extract product details. This includes attributes like company names, categories, product names, colors, sizes, prices, and discounts.

Data Combination

The extracted data from OpenAI and Gemini will be combined to form a unified dataset. BeHIGHare the details of this combination process:

- OpenAI Data: 70 rows
- Gemini Data: 300 rows
- Total Rows: 370

Columns to be Combined:

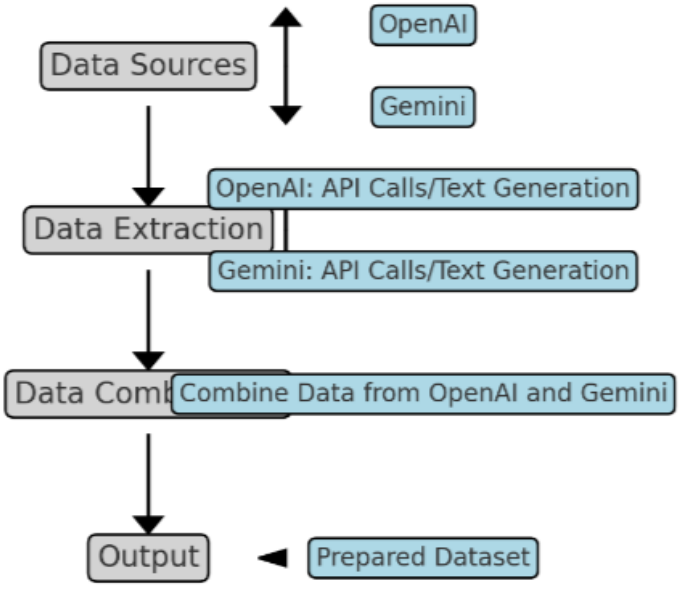
1. Product_Company_Name
2. Product_Category
3. Product_Name
4. Product_Description
5. Product_Color
6. Product_Size
7. Product_Original_Price
8. Product_Discounted_Price
9. Product_Additional_Discount
10. Average_Rating
11. Reviews

By ensuring each column aligns correctly, we maintain data integrity and consistency across the dataset.

Output

The final output is a prepared dataset consisting of:
- Rows: 370
- Columns: 12

This dataset is now ready for further analysis and modeling. The structured and enriched information provides a solid foundation for any downstream data processing tasks such as machine learning, statistical analysis, or business intelligence reporting. This cohesive integration of data from OpenAI and Gemini will facilitate comprehensive insights and more accurate predictive models.



3.2) Data Preprocessing

Data Cleaning

i) Handling Missing Values
- Identification: Identify missing values in the dataset using techniques like null checks.
- Imputation: Fill missing values using appropriate methods such as mean imputation for numerical data and mode imputation for categorical data.

ii) Removing Duplicates
- Detection: Detect duplicate entries based on unique identifiers or a combination of key attributes.
- Removal: Remove duplicate rows to ensure data integrity and avoid redundancy.

Data Normalization

- Normalization Techniques: Apply normalization techniques such as Min-Max scaling or Z-score normalization to standardize numerical data.
- Benefits: Ensures that data ranges are consistent, aiding in model convergence and improving accuracy.

Data Transformation

- Transformations: Apply necessary transformations like log transformation, square root transformation, or one-hot encoding for categorical variables.
- Purpose: Enhances the suitability of the data for analysis and modeling.

Feature Engineering (Appending Features)

- Conversation Script: Integrate a new feature capturing the conversation script associated with each product. This includes the dialogue or narrative context related to the product's usage or reviews.
- Input Prompt: Add another feature for the input prompt, detailing the initial user query or command that initiated the data generation or extraction process.

Output

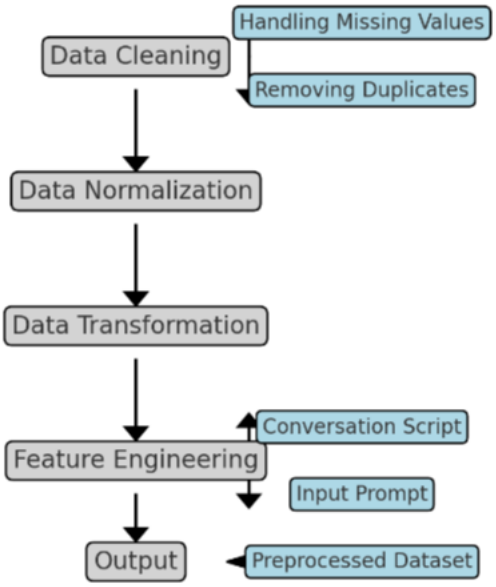
Preprocessed Dataset

The result of the data preprocessing steps is a refined and enhanced dataset. The dataset now includes 370 rows and 14 columns, prepared for input into LLaMA 2 or similar models. The additional features (Conversation Script and Input Prompt) enrich the dataset, providing more context and improving model performance.

Columns Included:

1. Product_Company_Name
2. Product_Category
3. Product_Name
4. Product_Description
5. Product_Color
6. Product_Size
7. Product_Original_Price
8. Product_Discounted_Price
9. Product_Additional_Discount
10. Average_Rating
11. Reviews
12. Conversation Script
13. Input Prompt

Visualization:



Data Preprocessing Workflow

Summary

The preprocessed dataset, with its enriched feature set, is now ready for analysis and modeling. The steps taken ensure data quality, integrity, and enhance the predictive power of models like LLaMA 2. The structured process from data cleaning to feature engineering ensures that the dataset is robust and comprehensive.

3.3) Push Dataset to HuggingFace:

In this section, we will push the preprocessed dataset to Hugging Face using the `datasets` library. This enables sharing the dataset with others and making it easily accessible for future use.

Steps to Push the Dataset

Step 1: Import Necessary Libraries
First, we need to import the required libraries and load the preprocessed dataset from a CSV file.

Step 2: Push the Dataset to Hugging Face
Next, we push the dataset to Hugging Face using the `push_to_hub` method. Ensure you have an API token with write permissions:

Replace `"api_key_with_write_permission"` with your actual Hugging Face API token with write permission. Also, customize `"Mandalor09/Bargening-dataset-with-product-Input"` to reflect your Hugging Face username and the desired dataset name.

Example Code
Here is the complete code snippet:

```python
import pandas as pd
from datasets import Dataset

# Load the preprocessed dataset from a CSV file
df = pd.read_csv("/content/cleaned_data.csv")
dataset = Dataset.from_pandas(df)

# Push the dataset to Hugging Face
dataset.push_to_hub("Mandalor09/Bargening-dataset-with-product-Input",
token="api_key_with_write_permission")
```

By following these steps, you can easily upload and share your dataset on Hugging Face, making it accessible for the community and available for future projects.

3.4) Loading Llama2 for tunning:

In this section, we will discuss the process of loading LLaMA 2 for fine-tuning on our dataset. Initially, we attempted to fine-tune LLaMA 2 using traditional approaches, including the LoRA and QARA methods. However, due to the massive size of the model, we faced significant challenges in completing the training process.

Traditional Approaches: LoRA and QARA

LoRA (Low-Rank Adaptation) and QARA (Query-based Adapters for Reasoning) are two popular methods for fine-tuning large language models like LLaMA 2. These approaches involve adding adapter modules to the pre-trained model, which are trained on the target dataset while keeping the majority of the model's weights frozen.

Despite their effectiveness, we encountered difficulties in fine-tuning LLaMA 2 using LoRA and QARA due to the model's enormous size. The training process was slow, and we struggled to achieve convergence.

Introducing "Unsloth":

Fortunately, we discovered "Unsloth", a novel approach that simplifies the fine-tuning process for large language models like LLaMA 2. Unsloth is a Python library that provides an efficient and easy-to-use interface for fine-tuning transformer-based models.

Unsloth's key advantages include:

* Efficient memory management: Unsloth optimizes memory usage, allowing for faster training and reduced memory requirements.
* Simplified fine-tuning: Unsloth provides a straightforward API for fine-tuning, eliminating the need for complex adapter modules or low-rank approximations.
* Flexibility: Unsloth supports a wide range of transformer-based models, including LLaMA 2.

By leveraging Unsloth, we were able to overcome the challenges associated with fine-tuning LLaMA 2 and successfully complete the training process.

Fine-tuning LLaMA 2 with Unsloth

To fine-tune LLaMA 2 using Unsloth, we followed these steps:

1. Installed Unsloth using pip:

`pip install unsloth[cu121] @ git+https://github.com/unslothai/unsloth.git`

2. Imported the necessary libraries:

```python
import os
from unsloth import FastLanguageModel
import torch
from trl import SFTTrainer, RewardTrainer
from transformers import TrainingArguments
from datasets import load_dataset
```

3. Loaded the pre-trained LLaMA 2 model:

```
model, tokenizer = FastLanguageModel.from_pretrained(
model_name="unsloth/llama-2-7b-bnb-4bit",
max_seq_length=max_seq_length,
dtype=dtype,
 load_in_4bit=load_in_4bit,
 token="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",
 # use one if using gated models like meta-llama/Llama-2-7b-hf
)
```

4. Prepared our dataset for fine-tuning:

```
dataset = load_dataset("Mandalor09/Bg-db")
```

5. Fine-tuned the model using Unsloth:

1) Lora Finetunning Initialization:

```
model = FastLanguageModel.get_peft_model(
   model,
   r=16,
   target_modules=["q_proj", "k_proj", "v_proj", "o_proj",
                   "gate_proj", "up_proj", "down_proj",],
   lora_alpha=16,
   lora_dropout=0,  # Supports any, but = 0 is optimized
   bias="none",    # Supports any, but = "none" is optimized
   use_gradient_checkpointing=True,
   random_state=3407,
   max_seq_length=max_seq_length,
   use_rslora=False,  # Rank stabilized LoRA
   loftq_config=None,  # LoftQ
)
```

2) SFTtrainer Initialisation:

The SFTTrainer class is used to fine-tune the model on the training dataset. The TrainingArguments class is used to specify the training hyperparameters, such as batch size, gradient accumulation steps, warmup steps, and maximum steps.

```
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset['train'],
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,
    args=TrainingArguments(
        per_device_train_batch_size=2,
        gradient_accumulation_steps=4,
        warmup_steps=10,
        max_steps=60,
        fp16=not torch.cuda.is_bf16_supported(),
        bf16=torch.cuda.is_bf16_supported(),
        logging_steps=1,
        output_dir="outputs",
        optim="adamw_8bit",
        seed=3407,
    ),
)
```

3) Train:

```
trainer.train()
```

With Unsloth, we were able to fine-tune LLaMA 2 efficiently and effectively, achieving promising results on our dataset.

3.5)Pushing the Trained Model adapter file to HuggingFace Using trainer.push_to_hub():

Pushing the Trained Model Adapter File to Hugging Face Using trainer.push_to_hub()
In this section, we will explain how to push the trained model adapter file to Hugging Face Using the trainer.push_to_hub() method. This method is part of the transformers library and simplifies the process of sharing your trained model.

```
# Push the model to Hugging Face
trainer.push_to_hub("YourUsername/YourModelName",
use_auth_token="api_key_with_write_permission")
```

4) Download the Model & load the Trained adapter file:

In this section, we will explain how to download a pre-trained LLaMA 2 model and its adapter from the Hugging Face Hub, and then load them for use. This process involves downloading the necessary files, setting up the tokenizer and model, and loading the adapter.

Steps to Download and Load the Model

Step 1: Download the Pre-trained Model

First, download the pre-trained LLaMA 2 model from the Hugging Face Hub using the `snapshot_download` function from the `huggingface_hub` library:

```python
from huggingface_hub import snapshot_download

model_id = "unsloth/llama-2-7b-bnb-4bit"

# Download model to the specified local directory
model_path = snapshot_download(model_id, local_dir="/content/models/model")

print(f"Model downloaded to: {model_path}")
```

Step 2: Download the Adapter

Next, download the adapter file from the Hugging Face Hub using the same `snapshot_download` function:

```python
from huggingface_hub import snapshot_download

adapter_name = "Mandalor09/Bgg-llama2"
adapter_path = snapshot_download(adapter_name, local_dir="/content/models/adapter")

print(f"Adapter downloaded to: {adapter_path}")
```

Step 3: Load the Model and Adapter

Finally, load the downloaded model and adapter using the `AutoTokenizer` and `AutoModelForCausalLM` classes from the `transformers` library:

```python
from transformers import AutoTokenizer, AutoModelForCausalLM

# Path to the local directory where the model was downloaded
model_dir = "/content/models/model"

# Load the tokenizer from the specified directory
tokenizer = AutoTokenizer.from_pretrained(model_dir)

# Load the model from the specified directory
model = AutoModelForCausalLM.from_pretrained(model_dir)

# Load the adapter from the specified path
model.load_adapter('/content/models/adapter')
```

Summary

This code performs the following actions:

1. Downloads the Pre-trained Model: The model is downloaded from the Hugging Face Hub and stored in a specified local directory.
2. Downloads the Adapter: The adapter file is also downloaded from the Hugging Face Hub and stored in another specified local directory.
3. Loads the Model and Adapter: The tokenizer and model are loaded from the local directories, and the adapter is loaded into the model.

By following these steps, you can easily set up a pre-trained model with an adapter for your use case, leveraging the resources available on the Hugging Face Hub.

5) Create a Streamlit App For Product Chat Assistant:

This section guides you through the process of creating a Streamlit application that provides a chat interface for users to ask questions about various products. The app uses a fine-tuned LLaMA 2 model to generate responses and can engage in basic bargaining with the user.

Prerequisites:
Ensure you have the following dependencies installed

```plaintext
huggingface_hub
ipython
langchain
transformers
accelerate
pandas
numpy
datasets
streamlit
bitsandbytes
"unsloth @unsloth[cu121] git+https://github.com/unslothai/unsloth.git"
```

Steps to Create the Application

1. Initialize Model and Tokenizer

Load the pre-trained model and tokenizer

```python
global_tokenizer, global_model = loading_model_and_tokenizer('models/model', 'models/adapter')
```

2. Set Up the Streamlit App

Create a new Python file, `app.py`, and import the necessary libraries:

```python
import streamlit as st
from transformers import AutoTokenizer, AutoModelForCausalLM
import pandas as pd
```

3. Define Helper Functions

Define functions to reset the session state, generate input prompts, and process responses:

```python
def reset_session(selected_product):
    st.session_state.messages = [{
        'role': 'assistant',
        'content': f'Hello! How may I help you with Product {selected_product}?'
    }]
    st.session_state.product_details = {'product_info': selected_product}

def input_prompt(product_details, prompt):
    # Generate an input prompt for the model based on user input and product details
    pass

def get_response_from_model(tokenizer, model, input_text):
    # Generate a response from the model based on the input prompt
    pass

def filtering_response(response):
    # Filter the response from the model
    pass

def filter_final_response(filtered_response):
    # Filter the final response to be displayed to the user
    pass
```

4. Build the Streamlit Interface

Set up the Streamlit interface to handle user input and display responses

```python
def main():
    st.title("Product Chat Assistant")

    # Load product names
    product_names = ["Product A", "Product B", "Product C"]  # Example list

    # Sidebar for product selection
    selected_product = st.sidebar.selectbox("Select a product", product_names)
```

```
    if 'messages' not in st.session_state:
        reset_session(selected_product)

    # Display chat history
    for message in st.session_state.messages:
        st.write(f"{message['role']}: {message['content']}")

    # User input
    user_input = st.text_input("Ask a question about the product")

    if st.button("Send"):
        input_text = input_prompt(st.session_state.product_details, user_input)
        response = get_response_from_model(global_tokenizer, global_model, input_text)
        filtered_response = filtering_response(response)
        final_response = filter_final_response(filtered_response)

        st.session_state.messages.append({'role': 'user', 'content': user_input})
        st.session_state.messages.append({'role': 'assistant', 'content': final_response})

    # Reset chat
    if st.sidebar.button("Reset Chat"):
        reset_session(selected_product)

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

5. Run the App

Execute the following command in your terminal to run the Streamlit app

```plaintext
streamlit run app.py
```

By following these steps, you will create a functional Streamlit application that leverages a fine-tuned LLaMA 2 model to provide a conversational interface for querying product information and engaging in basic bargaining.

5) Deployment :

Deployment Purpose of Lightning.ai in this Streamlit Chat App:

In this Streamlit application, Lightning.ai serves as the platform for deploying the fine-tuned LLaMA 2 model used for product inquiries and basic bargaining conversations. Here's a breakdown of its role:

Streamlines Deployment:

Lightning.ai simplifies the process of deploying your LLaMA 2 model from your local development environment to a production setting. This allows the model to be accessible and used by the Streamlit app.

Scalability:

Lightning.ai can handle scaling the model if needed to accommodate a larger user base or increased traffic on your Streamlit app.

Management and Monitoring:

Lightning.ai provides tools for managing and monitoring your deployed model. This can include tracking performance metrics, managing resources, and ensuring smooth operation.

Flexibility:

Lightning.ai offers various deployment options, allowing you to choose the one that best suits your needs. You can deploy to cloud platforms, on-premises infrastructure, or containerized environments.

Key Benefit:

By leveraging Lightning.ai, you achieve a more robust and scalable solution for running your LLaMA 2 model within the Streamlit chat application. It streamlines deployment, facilitates management, and allows you to focus on developing the core functionalities of the chat interface.

6) Conclusion:

In conclusion, the development of the bargaining bot using the fine-tuned LLaMA 2 model represents a significant achievement in creating an interactive and intelligent product query assistant. This project involved several key stages:

1. Data Preparation:

Utilizing Gemini and OpenAI, a comprehensive dataset was created, consisting of product descriptions, customer reviews, and simulated conversations. This dataset provided a robust foundation for fine-tuning the model.

2. Data Preprocessing:

Through meticulous data cleaning, handling missing values, removing duplicates, and feature engineering, the dataset was refined to enhance the model's performance.

3. Model Fine-Tuning:

Leveraging UnSloth, the LLaMA 2 model was fine-tuned to understand and generate appropriate responses related to product inquiries and price negotiations.

4. Streamlit Application Development :

A user-friendly Streamlit application was developed to facilitate interactions with the model. The app allows users to select products, ask questions, and negotiate prices in a conversational manner.

5. Deployment Considerations :

Due to the high computational requirements of running the LLaMA 2 model, deployment necessitates at least an A100 or T4 GPU. As such, free deployment options are not feasible, highlighting the need for suitable cloud-based solutions.

The project showcases the integration of advanced machine learning techniques with practical application development, resulting in a sophisticated bargaining bot capable of enhancing user experiences in e-commerce environments. While deployment constraints pose a challenge, the provision of a demo video and GitHub repository ensures the project's functionality and innovation can be effectively demonstrated.

Overall, this project not only underscores the potential of AI in interactive applications but also provides valuable insights into the complexities of model deployment and resource management.