Blueprint: Private AI Stack with LM Studio for Testing

Summary

This document is designed specifically for first-time users of LM Studio. It provides a comprehensive guide, outlining each step necessary to set up a functional working environment. The document refers to this setup as "Blueprint" and ensures users can easily follow the instructions. It aims to simplify the process, making it accessible for beginners.

LM Studio Setup

I. Prerequisites:

- Windows 10 or 11 (64-bit)
- Python 3.7 or later (Download from https://www.python.org/downloads/)
- NVIDIA or AMD GPU recommended (for faster training)

II. Installation:

- Download LM Studio: https://huggingface.co/lmstudio-ai (Choose the appropriate Windows installer)
- Double-click the installer and follow the on-screen instructions.

III. Verification:

- Open an LM Studio.
- If successful, it should display the LM Studio version information.

Model Selection

LM Studio supports various Large Language Models (LLMs). Here's a suggestion for your tech support chatbot:
Model: TheBloke Mistral Instruct 7B, Lava graphic, Lamma3

Download Model

- Open the LM Studio application.
- Click on "Models" on the left menu.
- Search for " TheBloke Mistral Instruct v0 1 7B"
- Click "Download Model."

Steps to test your prompts

- Go to the local server option in the left menu, click on Start the server, and select the Mistral 7B V01 model.
- Go to Chat and Enter the prompt to test. eg: what is the color of the sun

You now have a functional LLM on LM Studio, go ahead and play around with it. Listed below are some more Models you can play around with.

Open source models you can experiment with

Mistral

- Model Size: 7 billion parameters
- Developed by: Anthropic
- Key Features: Excels at open-ended generation tasks like writing stories, code generation, and question answering. Trained on a diverse corpus of web data. 1 Strengths: Strong performance on benchmarks like MMLU, good at multi-task learning.
- Weaknesses: Smaller model size compared to others, may struggle with more complex tasks.

Mixtral

- Model Size: 12 billion parameters
- Developed by: Anthropic
- Key Features: Combines the capabilities of Mistral with multimodal understanding, allowing it to process and generate images, text, and other data types. 1
- Strengths: Multimodal capabilities, good at tasks involving visual and textual data.
- Weaknesses: Larger model size requires more computational resources.

Llama 3

- Model Size: 60 billion parameters
- Developed by: Meta AI
- Key Features: Trained on a large and diverse corpus, excels at open-ended generation, question answering, and analysis tasks. 2
- Strengths: Large model size, strong performance on benchmarks like MMLU and ScienceQA.

Bloom

- Model Size: 176 billion parameters
- Developed by: Hugging Face and BigScience
- Key Features: Largest open multilingual language model, trained on data from 46 languages. 3
- Strengths: Excels at multilingual tasks, strong performance on benchmarks like XGLUE.
- Weaknesses: Massive model size requires substantial computational resources.

GPT-NeoX-20B

- Model Size: 20 billion parameters
- Developed by: EleutherAI
- Key Features: Trained on a filtered subset of web data, focused on improving safety and truthfulness. 4
- Strengths: Good balance of performance and resource requirements, emphasis on safety.
- Weaknesses: Smaller model size may limit capabilities compared to larger models. 4

MS Phi3

- Model Size: 3 billion parameters
- Developed by: Microsoft
- Key Features: Trained on a curated dataset focused on truthfulness and safety. Designed for open-domain question answering and dialogue. 5
- Strengths: Emphasis on safety and truthfulness, good for open-ended QA.
- Weaknesses: Relatively small model size limits capabilities.

GROK

- Model Size: 280 billion parameters
- Developed by: DeepMind
- Key Features: One of the largest open-source language models. Trained on a filtered dataset to improve truthfulness. 5

- Strengths: Massive scale enables strong performance on many NLP tasks.
- Weaknesses: Requires immense computational resources, may still exhibit biases.

Claude

- Model Size: 100 billion parameters
- Developed by: Anthropic
- Key Features: Trained using constitutional AI principles to be safe and truthful. Excels at analysis, coding, and open-ended tasks. 5
- Strengths: Strong analytical capabilities, emphasis on safety and truthfulness.
- Weaknesses: Large model size is computationally intensive.

GEMMA

- Model Size: 339 billion parameters
- Developed by: Google
- Key Features: One of the largest open multimodal models, can process images, text, and other data. 5
- Strengths: Multimodal capabilities, massive scale enables high performance.
- Weaknesses: Extremely computationally expensive, may have biases from training data.

This covers a wide range of powerful open-source language models with different capabilities, strengths, and resource requirements to consider for various use cases. 12345

Sources:

- Top 10 Hugging Face Models for TensorFlow - SabrePC
- Hugging Face – The AI community building the future.
- Hugging Face Pre-trained Models: Find the Best One for Your Task
- Models - Hugging Face
- 8 Top Open-Source LLMs for 2024 and Their Uses - DataCamp

If you are planning to setup custom scrips to interact with the LLM here are the instructions for setting up an LLM RAG Chatbot with Lang Chain using VS Code

Step1:
i. Create a New Python Project:

```
#mkdir langchain_chatbot
#cd langchain_chatbot
```

ii. Set Up a Virtual Environment:

```
#python -m venv venv
#.\venv\Scripts\activate
```

iii. Install Required Libraries

```
#python -m pip install langchain==0.1.0 openai==1.7.2 langchain-openai==0.0.2
langchain-community==0.0.12 langchainhub==0.1.14 python-dotenv
```

iv. Create the necessary directories and files in VS Code:

data/

- reviews.csv

langchain_intro/

- Chatbot.py
- create_retriever.py
- tools.py .env

.env

Step2:

i. Add OpenAI below API Key to .env File:

```
LMSTUDIO_API_KEY=not-needed
```

ii.Open chatbot.py in VS Code. Add the below code:

```
import dotenv
import os
import requests

# Load environment variables from .env file
dotenv.load_dotenv()

# Set the base URL for the local LM Studio server
api_base_url = "http://localhost:1234/v1"
api_key = os.getenv("LMSTUDIO_API_KEY")

# Function to get chat response from the local LM Studio server
def get_chat_response(question):
url = f"{api_base_url}/chat/completions"
 headers = {
 "Authorization": f"Bearer {api_key}",
 "Content-Type": "application/json"
 }
 payload = {
 "messages": [
 {"role": "system", "content": "You are an assistant knowledgeable about general
information. Provide concise and accurate answers."},
 {"role": "user", "content": question}
 ],
 "temperature": 0.7,
 "max_tokens": 150
 }
 response = requests.post(url, headers=headers, json=payload)
 try:
response_data = response.json()
 if "choices" in response_data:
 return response_data["choices"][0]["message"]["content"].strip()
 else:
print("Response did not contain 'choices':", response_data)
 return "Error: The response from the server did not contain the expected format."
except ValueError:
 return "Error: Unable to parse response from server."
```

```
# Main function to prompt user for input and display chatbot response
if __name__ == "__main__":
 while True:
 question = input("Enter your query (or type 'exit' to quit): ")
 if question.lower() == 'exit':
 print("Goodbye!")
 break
 response = get_chat_response(question)
print("Chatbot response:", response)
```

iii. Open create_retriever.py in VS Code Add the code:

```
import dotenv
import os
import requests
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

# Load environment variables from .env file
dotenv.load_dotenv()

# Set the base URL for the local LM Studio server
api_base_url = "http://localhost:1234/v1"
api_key = os.getenv("LMSTUDIO_API_KEY")

# Set up ChromaDB
REVIEWS_CHROMA_PATH = "chroma_data/"
reviews_vector_db = Chroma(persist_directory=REVIEWS_CHROMA_PATH,
embedding_function=OpenAIEmbeddings())
# Function to get chat response from the local LM Studio server
def get_chat_response(question):
relevant_docs = reviews_vector_db.similarity_search(question, k=3)
 context = "\n".join([doc.page_content for doc in relevant_docs])
full_prompt = f"Context: {context}\n\nQuestion: {question}"

url = f"{api_base_url}/chat/completions"
 headers = {
 "Authorization": f"Bearer {api_key}",
 "Content-Type": "application/json"
 }
```

```
payload = {
 "messages": [
 {"role": "system", "content": "You are an assistant knowledgeable about
healthcare. Only answer healthcare-related questions."},
 {"role": "user", "content": question},
 {"role": "system", "content": f"Context: {context}"}
 ],
 "temperature": 0.7,
 "max_tokens": 150
 }
 response = requests.post(url, headers=headers, json=payload)
 if response.status_code == 200:
 return response.json()["choices"][0]["message"]["content"].strip()
 else:
 return f"Error: {response.status_code}, {response.text}"

# Test the chatbot
if __name__ == "__main__":
 question = "Has anyone complained about communication with the hospital
staff?"
 print(get_chat_response(question))
```

iv. To Test the chatbot Run the Below Command:

```
#python langchain_intro/chatbot.py
```

(Topics coming soon)

AI Personality for Tech Support

Training Data:

1. Collect transcripts from past tech support interactions (chat logs, emails)
2. Include relevant technical documentation and FAQs.
3. Define your desired chatbot personality (e.g., friendly, informative, professional).

Determine the Chatbot's Target Purpose & Capabilities:

1. Understand the purpose of your chatbot. Will it provide medical recommendations, answer health-related questions, or assist with appointment scheduling?
2. Define the chatbot's capabilities, such as diagnosing diseases, suggesting treatments, or providing general health information.

Collect Relevant Data:

1. Question-Answer Datasets:

   i. Gather datasets containing medical questions and their corresponding answers. Look for open-source datasets related to health care.

   ii. Examples include:

- AmbigQA: A question-answering dataset with disambiguated questions1.
- CommonsenseQA: A multiple-choice dataset that requires common sense knowledge1.
- Cornell Movie-Dialogs Corpus: Conversations from movies1.
- The Ubuntu Dialogue Corpus: Multi-turn dialogues1.
- Consider using customer support logs, social media dialogues, and other relevant sources2.

Categorize and Annotate the Data:

1. Organize the collected data into categories based on medical topics (e.g., symptoms, treatments, medications).
2. Annotate the data with relevant labels (e.g., intent labels, disease names).

Balance the Data:

1. Ensure a balanced representation of different medical conditions and scenarios.
2. Avoid bias by including diverse examples.

Choose the Right Architecture:

1. Decide on the AI model architecture. Common choices include:
   i. Machine Learning (ML): Use ML algorithms (e.g., logistic regression, decision trees) for simpler tasks.
   ii. Deep Learning: Consider neural networks (e.g., recurrent neural networks, transformers) for more complex tasks.

Develop a Robust NLP Model:

1. Implement Natural Language Processing (NLP) techniques.
2. Train the model to understand medical terminology, context, and user queries.

Continual Learning:

1. Regularly update the model with new data to keep it up-to-date.
2. Monitor user interactions and refine the chatbot's responses.

Test the Dataset and Model:

1. Evaluate the dataset by testing the chatbot's responses.
2. Use metrics (e.g., accuracy, F1 score) to assess performance.

Persona Development:

Define your desired chatbot personality (e.g., friendly, informative, professional). Craft responses that embody that personality (use positive reinforcement, acknowledge user frustration)

Desired Personality Traits:

a. Friendly: Make the chatbot approachable and warm.
b. Informative: Provide accurate and helpful information.
c. Professional: Maintain a respectful and knowledgeable tone.

Crafting Responses:

a. Positive Reinforcement:
   i. Use encouraging language to motivate users:
   - "Great job seeking medical advice!"
   - "You're taking the right steps by asking questions."
   - "I appreciate your proactive approach."

Acknowledging Frustration:

a. When users express frustration or concern, empathize and offer support:

i. “I understand this can be overwhelming. Let’s explore your options.”

ii. “Thank you for your patience. Let’s find a solution together.”

iii. “I’m here to assist you through this process.”

Integration:

LM Studio offers various functionalities for integrating your chatbot into your existing platform (API, web interface). Refer to the documentation for specific instructions.

API Integration:

i. What is it? An API (Application Programming Interface) allows your chatbot to communicate with other software applications.

ii. How to Use It:

- Endpoint: LM Studio provides an API endpoint where you can send user queries.
- Request Format: Typically, you’ll send a POST request with the user’s message.
- Response Format: The API will return the chatbot’s response.

iii. Advantages:

- Real-time interaction.
- Seamless integration with your application.

Web Interface Integration:

i. What is it? You can embed the chatbot directly into your web application.

ii. How to Use It:

- Embed Code: LM Studio provides code snippets that you can include in your HTML.
- Customization: Customize the chatbot’s appearance (colors, fonts, etc.).
- Event Handling: Handle user interactions (e.g., button clicks) via JavaScript.

iii. Advantages:

- User-friendly interface.
- No need for users to leave your website.

Documentation:

i. Refer to the official LM Studio documentation for detailed instructions on both API and web interface integration.

ii. Follow the step-by-step guides to set up your chatbot.

Optimization Techniques

- Prompt Engineering: Craft clear and concise prompts that guide the model towards generating relevant and informative responses.
- Temperature: Adjust the temperature parameter to control the creativity and randomness of the model's responses. A lower temperature leads to more conservative and factual responses.
- Top-k Sampling: This technique limits the model's output to the top k most likely tokens, improving response coherence.

Prompt Engineering:

i. What is it? Prompt engineering involves crafting effective prompts to guide pre-trained models. It's about designing clear and concise input prompts that lead the model toward relevant and informative responses.

ii. Why is it important? Well-crafted prompts can significantly influence the quality of generated output. They set the context and steer the model's attention.

iii. Tips for Effective Prompt Engineering:

- Be specific: Clearly state the desired task or question.
- Include relevant keywords: Use terms related to the topic you want the model to address.
- Avoid ambiguity: Make sure the prompt leaves no room for misinterpretation.

Temperature:

i. What is it? The temperature parameter controls the creativity and randomness of the model's responses during generation.

ii. How it works:

- High temperature (e.g., 1.0): Leads to more diverse and creative outputs. The model explores different possibilities.
- Low temperature (e.g., 0.2): Results in more conservative and factual responses. The model sticks to what it knows.

iii. Choosing the right temperature:

- For informative and precise answers, use a lower temperature.
- For creative or exploratory responses, opt for a higher temperature.

Top-k Sampling:

i. What is it? This technique limits the model's output to the top k most likely tokens at each generation step.

ii. How it works:

- The model ranks all possible tokens based on their probabilities.
- It selects the top k tokens (where k is a predefined value) and samples from them.

Advantages:

i. Improves response coherence by avoiding unlikely or noisy tokens.

ii. Helps prevent the model from generating gibberish.

Additional Info:

1. **Robust Error Handling:**

- Implement robust error-handling mechanisms to gracefully manage unexpected inputs or errors encountered during interactions.
- Provide informative error messages to users and log error details for troubleshooting purposes.

2. **Compliance and Regulation:**

- Ensure compliance with relevant regulations and standards governing the handling of medical information and personal data (e.g., GDPR, HIPAA).
- Implement measures to protect user privacy and secure sensitive information transmitted during interactions.