Project Report: AI Chatbot for Mental Health Support

A Guided AI/ML Project

July 29, 2025

Abstract

This report details the development of a conversational AI agent designed to provide empathetic emotional support. The project encompasses the entire machine learning lifecycle, from environment setup and model selection to fine-tuning, backend API development, and frontend user interface creation. A pre-trained language model, microsoft/DialoGPT-small, was successfully fine-tuned on the empathetic_dialogues dataset to adapt its conversational style. The project addresses real-world development challenges, including model instability and hardware constraints, culminating in a locally deployed application powered by a custom AI model.

1 Project Overview

1.1 Objective

The primary goal of this project was to develop a conversational AI designed to provide basic, empathetic emotional support. The project followed a structured, hands-on learning path, beginning with foundational concepts and culminating in a deployed application. The core technical achievement was to fine-tune a pre-trained language model on a specialized dataset to adapt its conversational style to be more supportive and understanding.

1.2 Core Technologies

- **Programming Language:** Python
- **AI/ML Libraries:** Hugging Face transformers, datasets, PyTorch
- **Backend API:** Flask
- **Frontend UI:** Streamlit
- **Environment:** Anaconda, Kaggle Notebooks (for GPU-based fine-tuning), VS Code (for local development)

2 Methodology and Development Phases

The project was executed in distinct phases, moving from environment setup and model training to application development and deployment.

2.1 Phase 1: Environment and Version Control

The project began by establishing a clean and isolated development environment using Anaconda.

1. **Virtual Environment:** A Conda environment named mental_health_chatbot_env was created to manage project-specific dependencies.
2. **Version Control:** A Git repository was initialized and linked to GitHub. This practice ensured that all code changes were tracked.

2.2 Phase 2: Model Fine-Tuning (Kaggle)

This was the most critical and challenging phase, conducted on Kaggle to leverage free GPU resources.

1. **Initial Model Selection:** The project explored microsoft/DialoGPT and facebook/blenderbot-400M-distill as potential base models. The final successful model was a fine-tuned version of microsoft/DialoGPT-small.
2. **Dataset:** The empathetic_dialogues dataset from Hugging Face was chosen for fine-tuning.
3. **Challenges & Resolutions:**
    - **Model Collapse:** Early training attempts resulted in the model producing nonsensical output. This was resolved by lowering the learning rate and introducing gradient clipping.
    - **Hardware Limitations:** The free-tier Kaggle environment had limited disk space, causing OSError: No space left on device. This was overcome by switching to a smaller model and optimizing the Trainer arguments to save only the single best model checkpoint (save_total_limit=1).
4. **Successful Fine-Tuning:** After several iterations, the model was successfully fine-tuned, demonstrating a clear shift towards coherent and empathetic responses.

2.3 Phase 3: Backend Development (API)

A backend service was created to host the custom model.

1. **Flask API:** A lightweight Flask web server was created in app.py.
2. **Endpoint:** A /chat endpoint was defined to accept POST requests.
3. **Logging:** A logging feature was implemented to save all conversations to a chat_log.log file.

2.4 Phase 4: Frontend Development (UI)

A user-friendly interface was built using Streamlit.

1. **Streamlit Application:** The UI was defined in frontend.py.
2. **State Management:** Streamlit's session_state was used to persist the conversation history.
3. **API Communication:** The frontend sends requests to the Flask backend and displays the response.

2.5 Phase 5: Local Deployment

The final step was to run the complete application on a local machine, with the backend and frontend running concurrently in separate terminals.

3 Final Application Code

Below are the complete, final source code files for the project.

3.1 app.py (Backend)

```
# app.py (Final Version with Logging)
import torch
from flask import Flask, request, jsonify
from transformers import AutoModelForCausalLM, AutoTokenizer
import logging # Import the logging library

# --- 1. SETUP LOGGING ---
# This will save logs to a file named 'chat_log.log'
logging.basicConfig(filename='chat_log.log', level=logging.INFO,
                    format='%(asctime)s - %(message)s')

# --- 2. SETUP MODEL ---
app = Flask(__name__)
print("Loading fine-tuned model...")
model_name = "./mental-health-chatbot-final"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
print("Model loaded successfully!")

# --- 3. API ENDPOINT ---
@app.route('/chat', methods=['POST'])
def chat():
    user_input = request.json.get('message')
    chat_history_ids_tensor = request.json.get('history')

    # Let the AI model handle the response
    if chat_history_ids_tensor:
        chat_history_ids = torch.tensor(chat_history_ids_tensor)
    else:
        chat_history_ids = None

    new_user_input_ids = tokenizer.encode(user_input + tokenizer.eos_token,
                                          return_tensors='pt')
    bot_input_ids = torch.cat([chat_history_ids, new_user_input_ids], dim=-1) \
                    if chat_history_ids is not None else new_user_input_ids
    attention_mask = torch.ones_like(bot_input_ids)

    chat_history_ids = model.generate(
        bot_input_ids, attention_mask=attention_mask, max_new_tokens=100,
        pad_token_id=tokenizer.eos_token_id, do_sample=True, top_k=50,
        temperature=0.75
    )

    response = tokenizer.decode(chat_history_ids[:, bot_input_ids.shape[-1]:][0],
                                skip_special_tokens=True)

    # Log the conversation
    logging.info(f"User: {user_input} | Bot: {response}")

```

```
    return jsonify(- 'response': response, 'history': chat_history_ids.tolist()
    ")

@app.route('/', methods=['GET'])
def health_check():
    return "API is running!"

if ``name`` == ```main```:
    app.run(host='0.0.0.0', port=8080)
```

Listing 1: File: app.py

3.2 frontend.py **(Frontend)**

```
# frontend.py
import streamlit as st
import requests

# --- Page Configuration ---
st.set_page_config(page_title="AI Chatbot", layout="centered")
st.title("        Mental Health Support Chatbot")
st.write("This is a safe space. I'm here to listen without judgment.")
st.write("---")

# --- Session State for History ---
if 'history' not in st.session_state:
    st.session_state['history'] = []
if 'chat_history_ids' not in st.session_state:
    st.session_state['chat_history_ids'] = None

# Local API URL
API_URL = "http://127.0.0.1:8080/chat"

# --- Display Chat History ---
for user_msg, bot_msg in st.session_state.history:
    with st.chat_message("user"):
        st.markdown(user_msg)
    with st.chat_message("assistant"):
        st.markdown(bot_msg)

# --- User Input Handling ---
prompt = st.chat_input("How are you feeling today?")

if prompt:
    with st.chat_message("user"):
        st.markdown(prompt)

    api_data = -
        'message': prompt,
        'history': st.session_state.chat_history_ids
    "

    with st.spinner("Thinking..."):
        try:
            response = requests.post(API_URL, json=api_data)
            response.raise_for_status()

            result = response.json()
            bot_response = result['response']

            with st.chat_message("assistant"):
                st.markdown(bot_response)
```

```
49
50              st.session_state.history.append((prompt, bot_response))
51              st.session_state.chat_history_ids = result['history']
52
53          except requests.exceptions.RequestException as e:
54              st.error(f"API Error: Could not connect to the backend.")
55          except Exception as e:
56              st.error(f"An unexpected error occurred: {e}")
```

Listing 2: File: frontend.py