Creating a Chatbot Using a Fine-Tuned Model

Ali Chouaib
10/21/2024

Table of Contents

Introduction

The purpose of this paper is for the reader to follow along in my journey of developing a chatbot that uses a fine tuned model.

I am currently working on this as a project for my Generative AI and Large Learning Models (LLM) class and I hope to add this to my professional resume. We have been assigned, at the beginning of the semester, a book and we must fine tune our pre-made model to the book that we were given.

The book of choice for me is Plato's "The Republic". Plato is one of the most well known and renowned philosophers and hopefully we can get some interesting results from this project.

I am going to be documenting my progress here and attempt to be as specific as I can. I will also include the source code we were provided at the end of this paper as well as any other resources we were given prior to starting. I will note all errors I experienced as well as steps I took to troubleshoot them.

I hope to be able to guide someone through this process if they are not familiar with this sort of thing, as well as have this be a paper to give myself a refresher to reread in the future.

** Disclaimer: This is my first time working on anything this advanced. I currently feel underqualified to create such a thing but I welcome it as a learning experience.

Provided Materials

Here is a list of the things we were provided:

➔ **Azure for Students Subscription**: Provided with $100 credits to use for the project.
➔ **Azure OpenAI Resource**: Access to GPT-35-Turbo for model deployment.
➔ **GitHub Repository**: Pre-configured repository for project deployment.

- ➔ **Azure Static Web App**: Pre-set up hosting environment for the web interface.
- ➔ **Documentation Links**: Instructions for setting up Azure OpenAI and Azure Static Web Apps, provided through Azure and GitHub.

Procedure

This section will be split into multiple parts.

Part One: Azure Resources Setup

1.) **Activate your Azure Student Subscription**
- The first thing you'll need to do is to create an account with Microsoft Azure and activate your student subscription to earn your free $100 worth of credits.
- Alternatively if you are not a student, you'll have to link bank information to Azure to pay for the plan of your choice.
- Everything we do in this report will be free using the Students plan subscription.

2.) **Create a resource**
- Go to the Azure portal and click “Create a resource”. In the search bar type in “Azure OpenAI”, search then hit “create”
- For Subscription choose “Azure for Students”
- Create a new resource group or use one you already have made
- Choose the region closest to you
- Name your resource whatever you’d like
- For Pricing Tier we chose Standard S0
- Carry on through Networks, Tags, and at section 4 press submit

3.) **Deploy your model**
- Go to Open AI Studio and choose the Deployments tab on the left side
- Click Deploy model
- Choose “Deploy Base Model” (For this project we chose gpt-35-turbo)
- Name and deploy your model

**** Make sure to note down the following:**
- Azure Api Key
- Azure Resource Name
- Azure Deployment Name

Part Two: Azure Static Web App

In this section we're going to be deploying the application in our "ui" folder using Azure Static Web Apps.

1.) **Create the static web app**
    - Go to the Azure Login Portal and click on "Create a resource" and search for and press create static web app
    - Enter your subscription and choose the resource group that we created in the previous step
    - In this guide we chose to go with the free plan
    - Sign in with github and link your repo to the web app, choose the main branch

2.) **Deployment**
    - After you create your web app it may need some time to complete deployment
    - A new actions workflow will be created in github for continuous deployment
    - Once deployment is complete, go to the Azure portal and access the web app resource, go to overview and visit your site
    - You should see that your app is live, although at this point, the chatbot should not work

Part Three: Configuring Environment Variables

1.) **Creating the .env.local file**
    - You can go to either github codespaces or you can go to your repo, either way enter the ui folder (In codespaces type "cd ui" into the terminal)
    - If you are in your repo, copy the .env.example file and rename it to .env.local
    - If you're in codespaces, in the terminal type "cp .env.example .env.local"

2.) **Setting up .env.local**
    - Remove the existing data from after the = sign on all three lines
    - Do not include quotation marks and do not use a space after the = sign, enter the values that you noted earlier when setting up Azure Resources

3.) **Setting up environment variables in Azure**
    - Go to your static web app resource and click on "Environment Variables" under the "Settings" section on the left hand menu

- You need to add three variables here, one for each of the values in .env.local
- I named each variable the exact name in the .env.local file, I don't believe this was necessary. You can name it as you'd like, BUT the values of those variables must match
- Be sure to press the "Apply" button at the bottom before doing anything else as it will wipe all of those variables you just set up if you don't

Part Four: Testing Locally

We are going to ensure your application works correctly on your local machine before deploying changes.

1.) **Install Dependencies**
- In codespaces, make sure you are in the ui directory in your terminal
- Run "npm install"
- If you are getting errors make sure you are in the correct directory, it needs to find the package.json file and install the packages to it

2.) **Run the Development Server**
- Again, be sure you're in the ui directory in your terminal
- Run "npm run dev"
- In the bottom right of your screen for about 5 seconds it'll ask you if you want to navigate to your website, ive been using that
- Alternatively, the local development server is usually on "http://localhost:3000"
- Go to the your site and test the functionality, you should be able to have a chat with the chatbot
- If you receive an error after you first prompt, don't worry, the issue lies within your models deployment, if there is a lot of load on Azures Servers, they will limit the amount of responses you can get per minute

Part Five: Deploy the App

1.) **Push your changes**

- Go to codespaces and be sure to commit your changes
- Go to github actions and you should see a new workflow pop up, let it load and if all went well it'll go through
- If it fails, which is something I went through, it is most likely because your web apps deployment api key in github does not match your actual key to fix this:
    - Go to github repo settings
    - Under security choose "secrets and variables"
    - Choose actions and you should see "AZURE_STATIC_WEB_APPS_API_TOKEN..."
    - Press edit and then go to your web app resource page on Azure
    - Go to overview and at the top you should see "Manage Deployment Token"
    - Choose that and reset the token, after it generates a new token for you, copy the value and go back to your github tab
    - Enter the new token into the value tab (Deleting the old one if there is one or just pasting it into the empty field)
    - Hit update secret then go to codespaces
    - Make a small change anywhere (I did it in the readme file) and recommit the code

2.) **Test the online web app**
- Through your Azure portal go to your web app resource
- Navigate to your site and test the chatbot
- It should work (I did not come across any errors at this step)
- If you are only getting one response before an error, there's a good chance it is Azure's model that you deployed limiting you because of traffic

Part Six: Prepare Data

1.) **Choose your book**
- The book I chose is Plato's "The Republic"

2.) **Clean Data**
- Remove any and all unnecessary information from the text to keep the chatbot as accurate as possible

Functions

The following section is about each function I've created. I will list them in groups depending on what they are for, in the order I created them, which also happens to be the order you'd use them.

Group One: Preparing your JSON

chunkText

```
import { RecursiveCharacterTextSplitter } from 'langchain/text_splitter';

export default async function chunkText(text, size = 5000, overlap = 1500)
 {
   // Split the text into manageable chunks
   const textSplitter = new RecursiveCharacterTextSplitter(
   {
     chunkSize: size,
     chunkOverlap: overlap,
   });

   const docs = await textSplitter.createDocuments([text]);

   const chunks = docs.map(doc => doc.pageContent);

   return chunks;

  }
```

- This function is used to break the document/text you provide into smaller, more manageable pieces.
- It takes in a parameter 'text' which should be your data.
- 'Size' and 'overlap' are static parameters that we set manually, size sets the token-size of each chunk, and overlap sets the amount of tokens each chunk overlaps the chunk before and after it
- Returns an array of strings (chunks)

embedChunks

```
export default async function embedChunks(chunks)
{

        const embedding = await model.embedDocuments(chunks);

        const embeddedChunks = chunks.map((chunk, index) => (
         {
                chapter: `Chapter ${index + 1}`,
                content: chunk,
                embedding: embedding[index],
         }));

         return embeddedChunks;
 }
```

- This function embeds each chunk and returns and returns each chunk as an object with 3 variables: Chapter, Content, and Embedding

processBook

```
export default async function processBook()
{
 var bookText = "";
await fs.readFile('book.txt','utf8').then(data => bookText = data);

 // Split the book text into chunks
const chunks = await chunkText(bookText);

 console.log(chunks.length);

// Embed the text chunks
const embeddedChunks = await embedChunks(chunks);
```

```
            await fs.writeFile('./TheRepublic.json', JSON.stringify(embeddedChunks,
            null, 2));
            }
```

- This function takes text, splits it using my chunkText function then embeds using the embedChunks function
- It then stores it to a JSON file

Group Two: Data Processing

embedUserQuery

```
export default async function embedUserQuery(query)
{

            const embedding = await model.embedQuery(query);

             return embedding;
 }
```

- Simple function to embed an end users query

cosineSimilarity

```
function cosineSimilarity(vectorA, vectorB)
{
```

```
const dotProduct = vectorA.reduce((acc, val, i) => acc + val * vectorB[i], 0);
const magnitudeA = Math.sqrt(vectorA.reduce((acc, val) => acc + val * val, 0));
const magnitudeB = Math.sqrt(vectorB.reduce((acc, val) => acc + val * val, 0));

return dotProduct / (magnitudeA * magnitudeB);
}
```

- This function performs operation to give any embedded piece of data a similarity score from -1 to 1

findBestMatch

```
export default function findBestMatch(queryEmbedding, chunks)
{
        let bestMatch = null;
        let bestScore = -1;
        let arr = [[]];
        arr = chunks;
        arr.forEach((chunk) =>
        {
                const score = cosineSimilarity(chunk.embedding ,
                queryEmbedding);

                 if(score > bestScore)
                {
                 bestScore = score;
                bestMatch = chunk;
                }
        });

        return bestMatch;
 }
```

- This function finds the best match between a queryEmbedding and an array of objects (chunks) in order to find the most similar chunk
- Returns the chunk with the best score

botSearch

```
export default async function botSearch(userQuery)
{
    // Function to list blobs in a container
    async function readBlob(containerName, blobName)
    {
        const containerClient =
        blobServiceClient.getContainerClient(containerName);
        const blobClient = containerClient.getBlobClient(blobName);
        try
        {
            const downloadBlockBlobResponse = await
            blobClient.download(0);
            const downloadedContent = await
            streamToString(downloadBlockBlobResponse.readableStr
            eamBody);
            const jsonContent = JSON.parse(downloadedContent);
            return jsonContent;
         }catch(error)
          {
            console.log(error);
          }
    }

    async function streamToString(readableStream)
    {
        return new Promise((resolve, reject) =>
        {
            const chunks = [];
```

```
                    readableStream.on('data', (data) =>
                    {
                            chunks.push(data.toString());
                    });
                    readableStream.on('end', () =>
                    {
                            resolve(chunks.join(''));
                    });
                    readableStream.on('error', reject);
                });
        }

        // Call the function to test it
        const json = await readBlob('republic', 'TheRepublic.json');

        // Use the loaded data to process the user query
        const queryEmbedding = await embedUserQuery(userQuery);

        const bestChunk = findBestMatch(queryEmbedding, json);
        if (bestChunk != null)
        {
                console.log('chapter: ' + bestChunk.chapter);
                return bestChunk.content;
        } else
        {
                console.log("No suitable chunk found.");
                return "I don't have enough information to answer that.";
        }
    }
```

- Within this function are a few blob helper functions
- It reads the JSON data from the blob storage, then downloads and parses it
- The user query is embedded using my embedUserQuery function
- findBestMatch is then called to find the chunk from the data most similar to the users input
- Returns the most similar chunk

route.ts

In this section we will briefly look at the route.ts file.

Code -

```
export const maxDuration = 30;

export const POST = async (request: Request) =>
{
   // Parse the request body
   const requestData = await request.json();
   console.log("Request data:", requestData);

   // Ensure AZURE_DEPLOYMENT environment variable is set
   if (!process.env.AZURE_DEPLOYMENT)
   {
     throw Error("AZURE_DEPLOYMENT environment variable must be
provided.");
   }

   requestData.messages.unshift(
   {
      role: "system",
      content: [
      {
         type: "text",
         text: `
          You are an AI assistant named "Donald Nalls".
            The person responsible for creating you is Ali Chouaib, he is   also
considered the author of you.
          All context provided to you when answering peoples question will be
from Plato's famous book "The Republic with commentary".
          Never mention how you obtain your information or that you are
referring to any context. Just provide the answer as naturally as possible.
         If you cannot answer based on the provided information, simply say
"Please try to keep your questions related to Plato's "The Republic" " or guide
Alex towards a different approach without mentioning the context.
        `,
      }
```

```
  ],
});

// Extract the query from requestData.messages
const query = requestData.messages.pop();
if (!query || !query.content || !query.content[0].text)
{
   throw Error("Invalid query structure.");
 }

// Process the user query using processBook function
const context = await botSearch(query.content[0].text);
console.log("ProcessBook result:", JSON.stringify(context));

const prompt = `
Use the following context to answer the user's question in your own words.
If they mention a book, they are referring to Plato's "The Republic".
If they seem to be having small talk with you, just respond naturally.
Context: ${context}
User Question: ${query.content[0].text}
Answer:
`;

// Add the result as a message from the assistant
requestData.messages.push({
  role: "user",
  content: [
    {
      type: "text",
      text: prompt,
    }
  ],
});

const response = await getEdgeRuntimeResponse({
  options: {
    model: azure(process.env.AZURE_DEPLOYMENT),
  },
  requestData,
  abortSignal: request.signal,
});

// Add the response from the assistant
requestData.messages.push({
    role: "assistant",
    content: [
     {
       type: "text",
       text: response.text,
     }
    ],
   });

   return response;
  };
```

- This file is where the POST request handler lives
- We set a system message here to give some minimal context to the bot
- We extract the users query (the last message from requestData.messages is treated as the user's query
- We call botSearch to to process their query
- Created a response prompt as a message from the user to requestData.messages so the bot can respond based off of the context that best fits
- We add the response prompt to the messages
- Finally we log and return the assistants response

Libraries

Here I will list libraries used for external functions (API calls)

- ai-sdk/azure
- assistant-ui/react/edge
- langchain/text_splitter
- langchain/openai
- fs/promises
- Url
- azure/storage-blob

Azure Resources

Here I will list all azure resources I used

- Azure Storage account (To store embeddings)
- Azure OpenAi (Resource to hold models)
- Azure Static Web App (Host of my chatbot)

Conclusion

This project helped me understand how you can use your own data to influence a chatbot. It helped a lot with concepts I struggled to learn theoretically in class by forcing me to do a hands on approach and troubleshoot any issues I came across.