A Customer Service Chatbot Using Python, Machine Learning, and Artificial Intelligence

Ty EBSEN
University of Arkansas Little Rock
Little Rock, AR 72204 USA

Richard S. SEGALL
Arkansas State University
State University, AR 72467 USA

Hyacinthe ABOUDJA
Oklahoma City University
Oklahoma City, OK 73106 USA

Daniel BERLEANT
University of Arkansas Little Rock
Little Rock, AR72204 USA

ABSTRACT

This report shows that with the most recent advancements in Artificial Intelligence (AI) and Natural Language Processing (NLP) using generative-pretrained transformers, we can develop robust AI applications to assist customer service departments with question answer systems. This paper addresses the question answering task using an OpenAI Application Programming Interface (API). This report examines how to create an AI question answering application from documents that generated correct answers to questions about those documents. We used two different approaches to create the question answering system. One was to use just the OpenAI API. The other was to use the LangChain framework and libraries. Both applications did answer questions correctly. LangChain used less code with a higher learning curve. The OpenAI API used more code and provided more detailed answers.

Keywords: Artificial Intelligence, Chatbot, Machine Learning, Natural Language Processing, Python

1. BACKGROUND

Southwest Power Pool (SPP) is a regional transmission organization with 112 members and a footprint of 552,000-square-miles in 17 states It operates more than 70,000 miles of high-voltage transmission lines in the Eastern Interconnection. SPP also operates a Day-Ahead energy market with 324 market participants. SPP hosts several APIs which require two-factor authentication to access. SPP's customer service department receives numerous requests (many asking the same questions) from members requesting assistance with the APIs. Many of the questions are in the documentation that SPP supplies members when they participate in SPPs energy market. The documentation is also available for public download on SPPs website.

This purpose of this project is to build a customer service question answering system that members can use to ask questions regarding SPPs APIs. The ultimate goal is to allow customer service representatives to increase their efficiency by avoiding the need to answer the same questions repeatedly.

This project was done using an OpenAI API. We implemented the task in two different ways to support a comparison. We used the OpenAI API directly and also used the LangChain framework. The LangChain framework used very little code to accomplish this task. However, a lot of the process was hidden in the background. The OpenAI approach used more code, but the answers were noticeably more detailed.

Quality Assurance (QA) is any systematic process of determining whether a product or service meets specified requirements. This report next looks at some of the related work in the QA task area, then discusses the data collection process, followed by the approaches taken to implement the QA system and a comparison between the two approaches.

2. RELATED WORK

A step-by-step tutorial on how to build your own AI Chatbot with the ChatGPT API was published by Sha (2023a). That article lists a few things to keep in mind when creating a chatbot. These include that (i) a chatbot can be built on any platform; and therefore (ii) a powerful computer is not needed to create a chatbot. Sha (2023a) used Python, OpenAI, and Gradio to build a chatbot. Gradio is notable as a convenient way to demo machine learning projects with a web interface. Chatbots, in

ISBN: 978-1-950492-78-7
ISSN: 2771-5914

https://doi.org/10.54808/IMCIC2024.01.148

particular, can help convert a larger body of information into short, focused, customized extracts (Berleant and Berghel, 1994).

Sha (2023a) provided the following steps to set up the development environment for chatbot development. Steps 1-3 provide instructions for downloading and installing Python on your local computer, adding Python.exe to the PATH environment variable and checking that the installation was successful. The remaining steps are summarized individually next.

Step 4 is to upgrade PIP (Package Installer for Python). PIP is installed as part of the normal installation. PIP is the Python package manager that allows the user to install Python libraries. It is a good idea to upgrade PIP once the installation is complete.

Step 5 is to install the OpenAI and Gradio libraries using PIP. OpenAI provides the libraries that are used to access OpenAI's API. The Gradio libraries are used to create a web interface.

Step 6 is to download and install a code editor. The guide uses Notepad++, a simple and general purpose editor.

Step 7 is to obtain an OpenAI API (Application Programming Interface) Key. To use the OpenAI API the developer must have an API key that is provided by OpenAI. There is more than one way to get the API. One can check the OpenAI web site for free offerings or, alternatively, purchase credits. Sha walks developers through the process of getting an API key from OpenAI.

Step 8 is where Sha (2023a) provides code showing how to get a chatbot up and running, using "gpt-3.5-turbo" for the model. The developer can save this code and run it with Python. The instructions provided give the location of the URL for the required page.

Building the chatbot. Sha (2023b) follows up with further instructions. These give a step-by-step guide to building an AI chatbot using ChatGPT and constitute a good example to how to consume documents that are used to create an AI chatbot with a custom knowledge base.

Step 1: As in Sha (2023a), the author states that a chatbot developer can build a chatbot on any platform and that the guide is meant for such users.

Step 2: The author points out that since an AI is being trained as part of the process, it is recommended to take advantage of a good GPU due to the computational demands of the training.

Step 3: The guide recommends that the data set be in the English language for the example used. However, OpenAI works well with other popular languages such as French, Spanish, German, etc.

Step 4: As in the earlier guide, the author walks through the process of downloading and setting up the development environment with Python, PIP, and Gradio. Some new libraries are used in this guide. These include GPT Index, PyPDF2, PyCryptodone, and LangChain.

GPTIndex (also known as LlamaIndex) allows connecting external data to the LLM (Large Language Model). The example in this article uses an older version of GPTIndex.

PyPDF2 and PyCryptodome are Python libraries that will allow using PDF documents as input to the project.

The Gradio library allows interaction with a Chatbot using a web interface.

Step 5: Downloading and installation of the desired code editor.

Step 6: Walk-through of the process of obtaining an API key from OpenAI.

Step 7: Training and creating the chatbot. This is where the chatbot developer begins using their own documents for the knowledge base. The "gpt-3.5-turbo" model was used because it is cheaper and faster than many other models.

The instructions are to create a local folder called docs which will store all the pdf files that will be used. Note that text files and CSV files can also be stored. There is also a way to add SQL database files and that process is explained in another article (https://twitter.com/LangChainAI/status/1635304794335363072).

The larger the corpus of documents, the longer it takes to process them. The amount of time also depends on the CPU (Central Processing Unit) and GPU (Graphics Processing Unit) resources devoted to the task.

3. PROJECT BASICS

The Python code for this project is this:

```
from gpt_index import
  SimpleDirectoryReader,
  GPTListIndex,
  GPTSimpleVectorIndex,
  LLMPredictor, PromptHelper
from langchain.chat_models import
  ChatOpenAI
import gradio as gr
```

```
import sys
import os

os.environ["OPENAI_API_KEY"]
  = 'Your API Key'

def construct_index
  (directory_path):
    max_input_size = 4096
    num_outputs = 512
    max_chunk_overlap = 20
    chunk_size_limit = 600

  prompt_helper =
    PromptHelper(max_input_size,
                 num_outputs,
                 max_chunk_overlap,
                 chunk_size_limit
                 =chunk_size_limit)

  llm_predictor =
    LLMPredictor
     (llm=ChatOpenAI
       (temperature=0.7,
        model_name="gpt-3.5-turbo",
        max_tokens=num_outputs))

  documents =
    SimpleDirectoryReader
      (directory_path).load_data()

  index = GPTSimpleVectorIndex
    (documents,
     llm_predictor=llm_predictor,
     prompt_helper=prompt_helper)

  index.save_to_disk('index.json')
  return index

def chatbot(input_text):
index =
GPTSimpleVectorIndex.load_from_disk
  ('index.json')
response = index.query
  (input_text,
   response_mode="compact")
return response.response

iface = gr.Interface
  (fn=chatbot,
   inputs=gr.components.Textbox
     (lines=7,
      label="Enter your text"),
   outputs="text",
   title
      ="Custom-trained AI Chatbot")

index = construct_index("docs")
iface.launch(share=True)
```

This code first produces a file called index.json using the user defined construct_index function with all of the files in the docs folder/directory.

PromptHelper is a GPT_index class that deals with Large Language Model (LLM) context window token limitations. PromptHelper calculates the available context size using the context window size of a LLM, reserve token space for the prompt template, and the output.

A Large Language Model (LLM) is a type of Artificial Intelligence (AI) algorithm that uses deep learning techniques and potentially massively large data sets to understand, summarize, generate and predict new content (Kerner, 2023).

The Chatbot function is used to take the question or the input text and search the json.index for the answer. The function returns the response. The iFace object is used to create the Gradio web interface that will accept a prompt and return an answer.

Obtaining an answer to a question. Shapiro (2022) notes that "There is a need for answering questions from arbitrary volumes of data." This is a general problem statement for which the solution does not need to leverage knowledge about the type of document (Ding et al. 2006) or understanding a specific domain (Ding et al. 2005). Shapiro refers to this as multi-document answering and uses the U.S. Supreme Court opinion on Dobbs vs. Jackson which is about 450,000 characters long as the data source. OpenAI did have an answering API but it has been deprecated due to a lack of use. Shapiro's Multi Document Answering code is at https://github.com/daveshap/MultiDocumentAnswering.

The build_index.py script creates an index.json file from an input.txt file. The input.txt file contains the Dobbs vs Jackson opinion. The build_index.py script splits the data into 4000-word chunks. For each chunk an embedding is created using the OpenAI 'text-similarity-ada-001' engine. One way to think of this is as a vector with the text followed by the language representation. All chunks are combined with its embedding and written out to the index.json file.

The answer_questions.py script is where the answers to our questions are generated. The script first does a vector search for the given question and searches for the answer in the index.json file. A similarity score is calculated which is the dot product of the question vector and the vectors that are searched in the index.json file. The results are sorted by similarity score in descending order and the top 20 are returned.

The top 20 results from the index search are then sent to the OpenAI completion API with the same question. The results are then summarized into a final answer.

Attention. Vaswani (2017) notes that "The self-attention mechanism is key to the transformer. The attention mechanism allows the model to weigh the importance of different elements (or tokens) in the input sequence when generating representations. This enables the model to focus on relevant parts of the sequence, capturing long-range dependencies and improving performance on tasks that require understanding of context." (Vaswani, 2017)

Using HuggingFace. HuggingFace (2023) discussed question answering and provides open-source machine learning libraries. The question answering course is a step-by-step guide to using the libraries to fine-tune a model. The guide uses the Stanford Question Answering Dataset (SQuAD) dataset released by Stanford University. The SQuAD dataset is based on questions about Wikipedia articles. This is an extractive question-answer guide where the answer to the question is extracted from a given context.

The dataset is loaded using the HuggingFace dataset library. A function is created to tokenize the questions and context, generate the sequence_ids, and locate the start and end of the context. The function is then applied to the train split of the dataset. The model, train_dataset, and validation dataset are used as inputs to the Trainer class in the Hugging Face transformers library. The fine-tuned model was then pushed to Hugging Face and then used in a pipeline to answer the questions. This process produced answers that were nearly all reasonable.

4. DATA COLLECTION

PDF (Portable Document Format) files that contain information related to making API calls using two-factor authentication to Southwest Power Pools APIs were downloaded from spp.org. These documents are available publicly on spp.org. The data could potentially be collected from the SPP customer service system, but that data is not publicly available and permission would have to be obtained to use it outside of SPP.

5. METHODOLOGY

This project was developed using OpenAI APIs. OpenAI is an Artificial Intelligence company that is part non-profit and part for profit that was founded in 2015. In 2020 OpenAI introduced GPT-3 (Generative Pre-trained Transformer) a large language model that was trained on large datasets from the internet that is aimed at question answering.

We researched different ways of developing this QA application, experimenting with using both the OpenAI APIs and the LangChain libraries. We developed both applications using Python. We developed the LangChain application using Google Colaboratory notebook. We developed the OpenAI API application locally. It was necessary to obtain an OpenAI API key to use in both applications.

The document "two-factor authentication technical specifications v1.3 20170908.pdf" was used for this project. It is possible to use multiple documents by copying them into the "docs" folder of the project.

5.1 LangChain

LangChain is a framework that is used to build AI applications. Although the code is much more condensed than the OpenAI version, the LangChain learning curve was much steeper. The results using LangChain were just as good as the OpenAI approach. This application was developed in a Google Colaboratory notebook. There are several Python libraries that must be install first: langchain, openai, pydf, tiktoken, and chromadb. One of the features of LangChain that helped greatly is that it is not necessary to do any conversion of the pdf files. The chatbot developer can simply set the location of the pdf files in the script and it reads each file in the directory, splits the files, and generates the embeddings.

Here is listing of code with comments noted using #:

```
#import the OS library used for
#reading the API key from a local
#file and setting the required
#environment variable
#OPENAI_API_KEY.
import os

#Chroma is a vector data store that
#allows us to store our embeddings.
from langchain.vectorstores
  import Chroma

#OpenAIEmbeddings is used to create
#the embedding for the documents as
#well as the question.
from langchain.embeddings.openai
  import OpenAIEmbeddings

#TextSplitter allows us to split
#our input data into chunks. We
#have to do this because the OpenAI
#models have limits to the number
#of tokens that can be passed to
#the API.  For exampe the ada-001
#model the max number of tokens is
#2,049.
```

```
from langchain.text_splitter
  import
    RecursiveCharacterTextSplitter

#This is the LLM that is used from
#OpenAI.
from langchain.llms import OpenAI

#This library is used for
#retrieving the answer
from langchain.chains
  import RetrievalQA

#These two libraries are used to
#read a directory and load all PDF
#files.
from langchain.document_loaders
  import PyPDFLoader
from langchain.document_loaders
  import DirectoryLoader

#open file function used to open
#the file that contains the
#OpenAI API KEY
def open_file(filepath):
  with open(filepath, 'r',
            encoding='utf-8')
  as infile:
    return infile.read()

#Sets the environment variable
#OPENAI_API_KEY
os.environ['OPENAI_API_KEY']
  = open_file('openaiapikey.txt')

#This code loads all pdf files that
#are in the docs directory
loader =
  DirectoryLoader
    ('./docs/',
     glob="./*.pdf",
     loader_cls=PyPDFLoader)
documents = loader.load()

#splitting the text into 1000 byte
#chunks.
text_splitter =
  RecursiveCharacterTextSplitter
    (chunk_size=1000,
     chunk_overlap=200)
texts =
  text_splitter.split_documents
    (documents)

#creates the embeddings and creates
#a Chroma docsearch object that
#will be used to retrieve the
#answer to the question.
embeddings = OpenAIEmbeddings()
docsearch =
  Chroma.from_documents
    (texts, embeddings)
qa = RetrievalQA.from_chain_type
  (llm=OpenAI(),
   chain_type="stuff",
   retriever
     =docsearch.as_retriever())

query = "What is required for SPP\
  two-factor authentication?”
qa.run(query)
```

5.2 OpenAI

The OpenAI code it more involved than the LangChain approach but it is easier to follow and understand what is happening.

pdf_2_txt.py: This Python script uses the PyPDF2 and OS libraries to read all the pdf files in a directory, convert them to text, and save them to a file call input_docs.txt. The input_docs.txt will be the input to the create_index.py script.

```
import PyPDF2
import os
pdf_path =
 '/Users/tyebsen/projects/docs/'
for filename in os.listdir(pdf_path):
  if filename.endswith('.pdf'):
     reader = PyPDF2.PdfReader
       (os.path.join(pdf_path,
                     filename))
     for i in range(len(reader.pages)):
       page = reader.pages[i]
       page = page.extract_text()
       page = " ".join(page.split())
       print(page)
       file1 =
         open(r"input_docs.txt","a")
       file1.writelines(page)
       file1.close()
```

Create_index.py: This Python script reads the input_docs.txt file, and breaks the data into 3,000-byte chunks. For each chunk, it generates the embedding, saving the text and its embedding in a json formatted file called index.json. This file is used by the qa.py Python script to answer questions about the pdf files. We use 3,000-byte chunks because it is safely under the model prompt limit of 4,000 bytes.

```
Import openai
import json
import textwrap

def open_file(filepath):
  with open(filepath, 'r',
            encoding='utf-8')
  as infile: return infile.read()
```

```
openai.api_key =
  open_file('openaiapikey.txt')

def gpt3_embedding
  (content,
   Engine
     ='text-embedding-ada-002'):
  response =
    openai.Embedding.create
      (input=content,engine=engine)
  vector = # this is a normal list
  response['data'][0]['embedding']
  return vector

if __name__ == '__main__':
  alltext =
    open_file('input_docs.txt')
  chunks =
    textwrap.wrap(alltext, 3000)
  result = list()
  for chunk in chunks:
    embedding=gpt3_embedding
      (chunk.encode
        (encoding=
          'ASCII',
           errors='ignore'
        ).decode()
      )
    info =
      {'content': chunk,
       'vector': embedding}
    print(info, '\n\n\n')
    result.append(info)
  with open('index.json', 'w')
  as outfile:
    json.dump(result, outfile,
              indent=2)
```

qa.py: This Python script is heavily based on Shapiro's multi-document project that scans all the documents with the same question and returns the top 20 results. Shapiro used a dot product to determine the score. We updated the script to use cosine similarity for the score. Cosine similarity is what OpenAI recommends. The final answer is generated from a detailed summary of the 20 results.

```
import openai
import json
import numpy as np
import textwrap
import re
from time import time,sleep
from numpy.linalg import norm

def open_file(filepath):
    with open(filepath, 'r',
              encoding='utf-8')
    as infile: return infile.read()

openai.api_key =
  open_file('openaiapikey.txt')

def gpt3_embedding
  (content,
   engine=
     'text-embedding-ada-002'):
  content = content.encode
    (encoding='ASCII',
             errors='ignore'
    ).decode()
  response =
    openai.Embedding.create
      (input=content,engine=engine)
  vector =   #this is a normal list
   response['data'][0]['embedding']
  return vector

def similarity(v1, v2):
# return cosine similarity
  return
    np.dot(v1,v2)/norm(v1)*norm(v2)

def search_index(text, data,
                 count=20):
  vector = gpt3_embedding(text)
  scores = list()
  for i in data:
    score = similarity(vector,
                       i['vector'])
    scores.append
      ({'content': i['content'],
        'score': score})
  ordered = sorted
    (scores,
     key=lambda d: d['score'],
     reverse=True)
  return ordered[0:count]

def gpt3_completion
  (prompt,
   engine='text-davinci-002',
   temp=0.6, top_p=1.0,
   tokens=2000, freq_pen=0.25,
   pres_pen=0.0, stop=['<<END>>']):
     max_retry = 5
     retry = 0
     prompt =
       prompt.encode
        (encoding='ASCII',
         errors='ignore'
        ).decode()
     while True:
       try:
         response =
           openai.Completion.create
             (engine=engine,
              prompt=prompt,
              temperature=temp,
```

```
            max_tokens=tokens,
            top_p=top_p,
            frequency_penalty=
              freq_pen,
            presence_penalty=
              pres_pen,
            stop=stop
           )
        text = response
          ['choices'][0]['text']
          .strip()
        text =
          re.sub('\s+', ' ', text)
        filename =
          '%s_gpt3.txt' % time()
        with open(
          'gpt3_logs/%s'
            % filename,
          'w')
        as outfile:
outfile.write('PROMPT:\n\n'
  + prompt +
  '\n\n==========\n\nRESPONSE:\n\n'
  + text)
        return text
      except Exception as oops:
        retry += 1
        if retry >= max_retry:
          return "GPT3 error: %s"
            % oops
        print
          ('Error communicating \
with OpenAI:',
           oops)
        sleep(1)

if __name__ == '__main__':
  with open('index.json', 'r')
  as infile:
    data = json.load(infile)
  while True:
    query = input("Enter your \
question here: ")
    results =
      search_index(query, data)
    answers = list()
      #answer the same question
      #for all returned chunks
    for result in results:
      prompt=f"Use the following \
passage to give a detailed answer \
to the question:\n\QUESTION: \
{query}\n\nPASSAGE: \
{result['content']}\n\nDETAILED \
ANSWER:"
      answer =
        gpt3_completion(prompt)
      answers.append(answer)
    #summarize the
    #answers together
    all_answers =
      '\n\n'.join(answers)
    chunks = textwrap.wrap
      (all_answers, 10000)
    final = list()
      for chunk in chunks:
        prompt =
f"Write a detailed summary of the \
following: {chunk}"
        summary =
          gpt3_completion(prompt)
        final.append(summary)
      print('\n\n=========\n\n',
            '\n\n'.join(final))
```

6. RESULTS

Both approaches generated reasonable answers. However, the OpenAI approach generated much more detailed answers. The OpenAI approach required much more code, but we believe it provides a better understanding of what is actually happening with OpenAI. In comparison, the LangChain framework hides a lot of what is going on behind the scenes.

We also scored each predicted answer with the actual answer from the documents using the cosine similarity calculation. The LangChain answers scored better against the actual answers than the OpenAI method. Table 1 shows the LangChain results.

7. CONCLUSIONS

This project shows that, using the latest advances in Artificial Intelligence (AI) and Natural Language Processing (NLP), a chatbot developer can produce a robust QA (Quality Assurance) system that can be used to assist customers. OpenAI was selected for use because is currently leading in the AI Natural Language Processing (NLP) field with generative-pretrained transformer models. OpenAI does charge a small fee for the use of their Application Programming Interfaces (APIs), but it was negligible for this project.

We also chose the LangChain framework because of its ability to consume multiple documents. Although a lot of the processing is done behind the scenes, it is a powerful set of libraries that can be used to build AI applications albeit with a bigger learning curve than the OpenAI API.

The OpenAI version needs some performance enhancements for a production system, as it otherwise can take too long to produce an answer. One thing to look at is lowering the number of results returned from the search_index function from 20. This could however possibly reduce the accuracy of the results. Further testing would need to be done. Further research could also be done

using vector stores like Chroma rather than a JSON (JavaScript Object Notation) formatted file. The LangChain version performs much better.

For a production system, we would probably recommend the LangChain version. The code would be easier to maintain, the performance better, and using multiple pdf files (or any other type of files) is easier since it is only necessary to drop them into a folder without having to do format conversion or other manipulation.

We conclude that Southwest Power Pool (SPP) and other organizations could benefit from an AI question answering chatbot system to assist users with their API questions. Additional performance, load, and accuracy testing would need to be done prior to production. User testing would also need to be done. Security would also be a consideration in a project like this, as hackers should not be able to use such a service to gain access to internal organization systems.

It is important to note that SPP is a non-profit organization. A project that costs less than a certain limit would not normally be subject to a cost-based analysis. Also, these types of projects would best be discussed in multiple working groups made up of SPP members and staff to determine the viability of the project, as could be recommended for other organizations as well.

To calculate the "people cost," the rate of $41.00 was used as it is published in the Department of Labor "Employer Costs for Employee Compensation – March 2023" report. The Open AI cost was priced at 12,000 transactions at approximately $0.00700 per transaction (https://gptforwork.com/tools/openai-chatgpt-api-pricing-calculator).

8. FUTURE DIRECTIONS

Additional research could be done in providing metrics for a production QA (Quality Assurance) system. These metrics would be used to determine how the QA system is performing based on metrics such as conversation length, number of conversations, number of unique users, human takeover rate, and others. Michelle Cyca describes these metrics in the article "Chatbot Analytics 101: Essential Metrics to Track" (Cyca, 2022).

More research could also be done with different OpenAI models. This project used the text-embedding-ada-002 engine for the embeddings and text-divinci-002 for the completions.

9. ACKNOWLEDGMENT

Publication of this work was supported in part by the National Science Foundation under Award No. OIA-1946391. The content reflects the views of the authors and not necessarily the NSF.

10. REFERENCES

Berleant, D. and Berghel, H. (1994). Customizing Information: Part 1, Getting what We Need, when We Need It. **Computer**, vol. 27, no. 9, pp. 96–98, https://ieeexplore.ieee.org/document/312053.

Cheng, R. (2021). Question Answering with Pretrained Transformers Using PyTorch. **Toward Data Science.** [Online] January 19, 2021. https://towardsdatascience.com/.

Cyca, M. (2022). Chatbot Analytics 101: Essential Metrics to Track. https://blog.hootsuite.com/chatbot-analytics/. [Online] September 21, 2022.

Ding, J., Hughes, L.M., Berleant, D., Fulmer, A.W., Wurtele, E.S. (2006). PubMed Assistant: A Biologist-Friendly Interface for Enhanced PubMed Search, **Bioinformatics**, vol. 22, issue 3, pp. 378–380, https://doi.org/10.1093/bioinformatics/bti821.

Ding, J., Viswanathan, K., Berleant, D., Hughes, L., Wurtele, E.S., Ashlock, D., Dickerson, J.A., Fulmer, A., Schnable, P.S. (2005). Using the Biological Taxonomy to Access Biological Literature with PathBinderH, **Bioinformatics**, vol. 21, issue 10, pp. 2560–2562, https://doi.org/10.1093/bioinformatics/bti381.

HuggingFace (2023). Question Answering. **Hugging Face**. [Online] 2023. https://huggingface.com.

Kerner, S.M. (2023). Large Language Model. Retrieved August 6, 2023 from https://www.techtarget.com/whatis/definition/large-language-model-LLM.

Khanna, C. (2021). Question Answering with a fine-tuned BERT. **Towards Data Science**. [Online] May 15, 2021. https://towardsdatascience.com/.

Sha, A. (2023a). How to Build Your Own AI Chatbot With ChatGPT API: A Step-by-Step Tutorial. beebom.com. [Online] June 19, 2023. https://beebom.com/how-build-own-ai-chatbot-with-chatgpt-api/.

Sha, A. (2023b). How to Train an AI Chatbot With Custom Knowledge Base Using ChatGPT API. beebom.com. [Online] June 14, 2023. https://beebom.com/how-train-ai-chatbot-custom-knowledge-base-chatgpt-api/.

Shapiro, D. (2022). Answer complex questions from an arbitrarily large set of documents with vector search and GPT-3. youtune.com. [Online] June 25, 2022. https://www.youtube.com/watch?v=es8e4SEuvV0.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L.. Gomez, A.N., Kaiser, L., Polosukhin, I. (2017). Attention Is All You Need. In: **NIPS'17**: **Proceedings of 31st International Conference on Neural Information Processing Systems**. Dec.2021. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Table 1: *LangChain Results*

Question	Answer	LangChain Answer	COSINE Similarity Score
1) What is required for SPP two-factor authentication?	In order to authenticate, users or applications accessing the User Interfaces (UIs) and APIs must present a valid x509 certificate and present a second authentication factor.	A valid x509 client Certificate Authority (CA) that is trusted by SPP and a login ID (Identification) and password.	0.849
2) What is the integrated marketplace two-factor authentication?	In order to authenticate, users or applications accessing the UIs and APIs must present a valid x509 certificate and present a second authentication factor.	The integrated marketplace two-factor authentication requires users to present a valid x509 certificate and a second authentication factor, such as a login ID and password.	0.933
3) What is the format of the request header?	The concatenated token, made up of the Request Created Timestamp, Request Nonce, and Hash-based message authentication code(HMAC) Sha512 Hash, concatenated together and separated by hyphens.	The request header consists of the Request Created Timestamp, Request Nonce, and HMAC Sha512 Hash, concatenated together and separated by hyphens.	0.954
4) What is the format of the token?	requestCreated + "-" + requestNonce + "-"	The token is a Nonce and HMAC Sha512 Hash, concatenated together and separated by hyphens.	0.823
5) What is the format of the timestamp?	yyyy-Mdd'T'HH:mm:ss'Z' where 'T' is the literal character T and 'Z' is the literal character Z.	The format of the timestamp is yyyy-MM-dd'T'HH:mm:ss'Z', with yyyy being the 4 digit year, MM being the 2 digit month (01-12), dd being the 2 digit day (01-31), HH being the 2 digit hour (00-23), mm being the 2 digit minute (00-59), and ss being the 2 digit second (00-50).	0.880