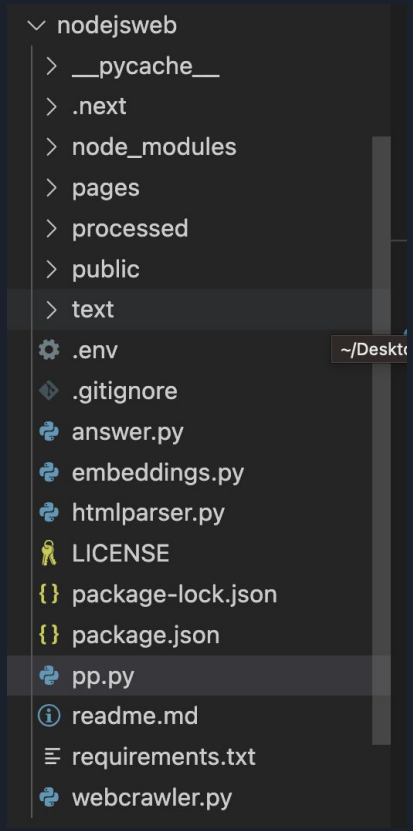
OpenAI API Interaction & Web Crawler

Table of Contents

Introduction

- This project comprises functionalities related to interaction with the OpenAI API and a web crawler. The codebase integrates Python scripts for backend processing and JavaScript for the frontend.

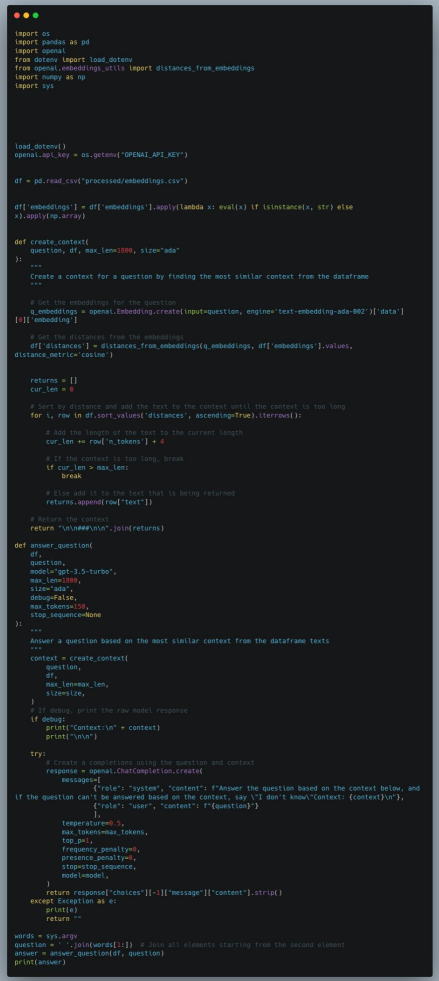


Python Scripts Overview

- **answer.py**: This script is the bridge to OpenAI. It uses a pre-processed embeddings dataset to interact with the OpenAI API and obtain relevant answers based on the given context.\
- **embeddings.py**: As the name suggests, this script is all about embeddings. Embeddings are generated and manipulated, which are then used to provide context for the questions posed to OpenAI.
- **htmlparser.py**: An integral part of the web crawling mechanism. This script extracts hyperlinks from HTML content, paving the way for the web crawler to traverse websites effectively.
- **webcrawler.py**: The heart of the web crawling mechanism. It's designed to crawl the openai.com domain, fetch web pages, and extract hyperlinks for further crawling.ers for AND Gate.

Answer.py

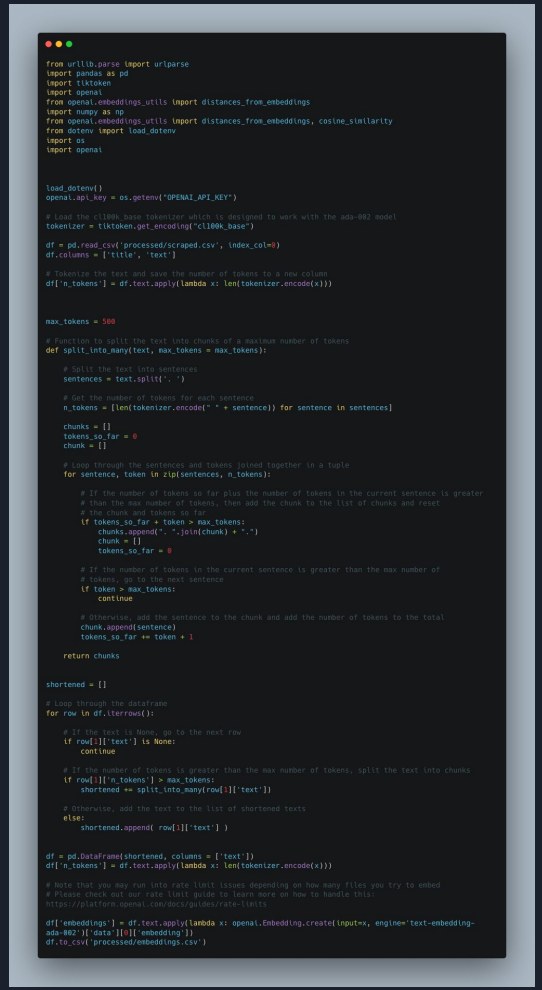
- This script serves as the main interface with the OpenAI API. It reads a pre-processed embeddings dataset from a CSV file and uses it to provide context when querying the OpenAI API. The resulting responses from the API are processed and returned based on this context.



```
import os
import pandas as pd
import openai
from dotenv import load_dotenv
from openai.embeddings_utils import distances_from_embeddings
import numpy as np
import sys




load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")


df = pd.read_csv("processed/embeddings.csv")


df['embeddings'] = df['embeddings'].apply(lambda x: eval(x) if isinstance(x, str) else
x).apply(np.array)


def create_context(
    question, df, max_len=1800, size="ada"
):
    """
    Create a context for a question by finding the most similar context from the dataframe
    """

    # Get the embeddings for the question
    q_embeddings = openai.Embedding.create(input=question, engine='text-embedding-ada-002')['data']
[0]['embedding']

    # Get the distances from the embeddings
    df['distances'] = distances_from_embeddings(q_embeddings, df['embeddings'].values,
distance_metric='cosine')


    returns = []
    cur_len = 0

    # Sort by distance and add the text to the context until the context is too long
    for i, row in df.sort_values('distances', ascending=True).iterrows():

        # Add the length of the text to the current length
        cur_len += row['n_tokens'] + 4

        # If the context is too long, break
        if cur_len > max_len:
            break

        # Else add it to the text that is being returned
        returns.append(row["text"])

    # Return the context
    return "\n\n###\n\n".join(returns)

def answer_question(
    df,
    question,
    model="gpt-3.5-turbo",
    max_len=1800,
    size="ada",
    debug=False,
    max_tokens=150,
    stop_sequence=None
):
    """
    Answer a question based on the most similar context from the dataframe texts
    """
    context = create_context(
        question,
        df,
        max_len=max_len,
        size=size,
    )
    # If debug, print the raw model response
    if debug:
        print("Context:\n" + context)
        print("\n\n")

    try:
        # Create a completions using the question and context
        response = openai.ChatCompletion.create(
            messages=[
                    {"role": "system", "content": f"Answer the question based on the context below, and
if the question can't be answered based on the context, say \"I don't know\"\"Context: {context}\n"},
                    {"role": "user", "content": f"{question}"}
                    ],
            temperature=0.5,
            max_tokens=max_tokens,
            top_p=1,
            frequency_penalty=0,
            presence_penalty=0,
            stop=stop_sequence,
            model=model,
        )
        return response["choices"][-1]["message"]["content"].strip()
    except Exception as e:
        print(e)
        return ""

words = sys.argv
question = ' '.join(words[1:])  # Join all elements starting from the second element
answer = answer_question(df, question)
print(answer)
```

embeddings.py

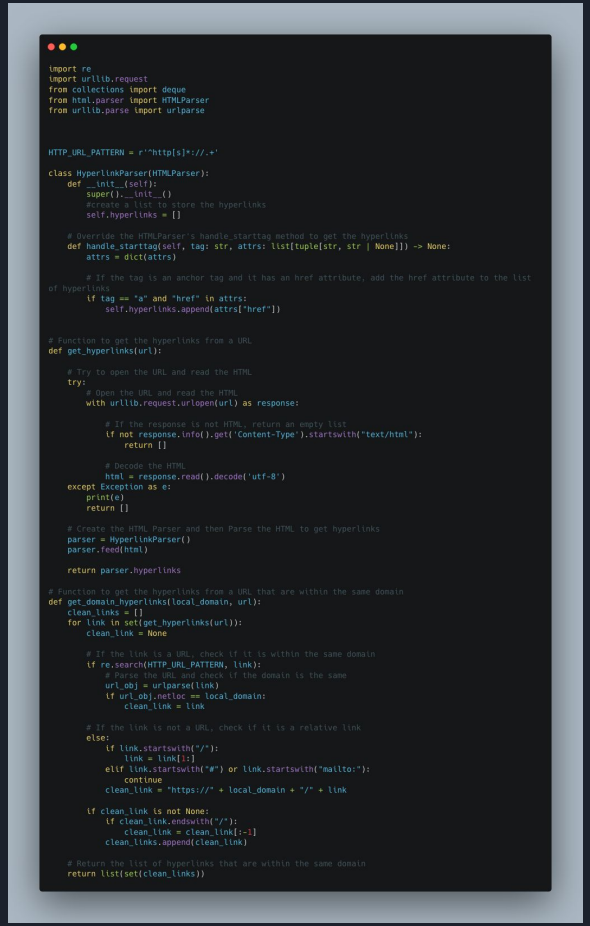
- The script sets up communication with the OpenAI API by loading the necessary API key from environment variables. Its primary function is to generate embeddings, which are vector representations of text data. After processing these embeddings, they are stored in a CSV file for subsequent use in the answer.py script.



```python
from urllib.parse import urlparse
import pandas as pd
import tiktoken
import openai
from openai.embeddings_utils import distances_from_embeddings
import numpy as np
from openai.embeddings_utils import distances_from_embeddings, cosine_similarity
from dotenv import load_dotenv
import os
import openai


load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# Load the cl100k_base tokenizer which is designed to work with the ada-002 model
tokenizer = tiktoken.get_encoding("cl100k_base")

df = pd.read_csv('processed/scraped.csv', index_col=0)
df.columns = ['title', 'text']

# Tokenize the text and save the number of tokens to a new column
df['n_tokens'] = df.text.apply(lambda x: len(tokenizer.encode(x)))


max_tokens = 500

# Function to split the text into chunks of a maximum number of tokens
def split_into_many(text, max_tokens = max_tokens):

    # Split the text into sentences
    sentences = text.split('. ')

    # Get the number of tokens for each sentence
    n_tokens = [len(tokenizer.encode(" " + sentence)) for sentence in sentences]

    chunks = []
    tokens_so_far = 0
    chunk = []

    # Loop through the sentences and tokens joined together in a tuple
    for sentence, token in zip(sentences, n_tokens):

        # If the number of tokens so far plus the number of tokens in the current sentence is greater
        # than the max number of tokens, then add the chunk to the list of chunks and reset
        # the chunk and tokens so far
        if tokens_so_far + token > max_tokens:
            chunks.append(". ".join(chunk) + ".")
            chunk = []
            tokens_so_far = 0

        # If the number of tokens in the current sentence is greater than the max number of
        # tokens, go to the next sentence
        if token > max_tokens:
            continue

        # Otherwise, add the sentence to the chunk and add the number of tokens to the total
        chunk.append(sentence)
        tokens_so_far += token + 1

    return chunks


shortened = []

# Loop through the dataframe
for row in df.iterrows():

    # If the text is None, go to the next row
    if row[1]['text'] is None:
        continue

    # If the number of tokens is greater than the max number of tokens, split the text into chunks
    if row[1]['n_tokens'] > max_tokens:
        shortened += split_into_many(row[1]['text'])

    # Otherwise, add the text to the list of shortened texts
    else:
        shortened.append( row[1]['text'] )


df = pd.DataFrame(shortened, columns = ['text'])
df['n_tokens'] = df.text.apply(lambda x: len(tokenizer.encode(x)))

# Note that you may run into rate limit issues depending on how many files you try to embed
# Please check out our rate limit guide to learn more on how to handle this:
https://platform.openai.com/docs/guides/rate-limits

df['embeddings'] = df.text.apply(lambda x: openai.Embedding.create(input=x, engine='text-embedding-
ada-002')['data'][0]['embedding'])
df.to_csv('processed/embeddings.csv')
```

htmlparser.py

- This script is an extension of the standard HTMLParser module. Its main objective is to parse provided HTML content meticulously. While doing so, it specifically looks for and extracts hyperlinks, which are then used by the web crawler for further navigation and data retrieval.



```python
import re
import urllib.request
from collections import deque
from html.parser import HTMLParser
from urllib.parse import urlparse



HTTP_URL_PATTERN = r'^http[s]*://.+'

class HyperlinkParser(HTMLParser):
    def __init__(self):
        super().__init__()
        #create a list to store the hyperlinks
        self.hyperlinks = []

    # Override the HTMLParser's handle_starttag method to get the hyperlinks
    def handle_starttag(self, tag: str, attrs: list[tuple[str, str | None]]) -> None:
        attrs = dict(attrs)

        # If the tag is an anchor tag and it has an href attribute, add the href attribute to the list
of hyperlinks
        if tag == "a" and "href" in attrs:
            self.hyperlinks.append(attrs["href"])


# Function to get the hyperlinks from a URL
def get_hyperlinks(url):

    # Try to open the URL and read the HTML
    try:
        # Open the URL and read the HTML
        with urllib.request.urlopen(url) as response:

            # If the response is not HTML, return an empty list
            if not response.info().get('Content-Type').startswith("text/html"):
                return []

            # Decode the HTML
            html = response.read().decode('utf-8')
    except Exception as e:
        print(e)
        return []

    # Create the HTML Parser and then Parse the HTML to get hyperlinks
    parser = HyperlinkParser()
    parser.feed(html)

    return parser.hyperlinks

# Function to get the hyperlinks from a URL that are within the same domain
def get_domain_hyperlinks(local_domain, url):
    clean_links = []
    for link in set(get_hyperlinks(url)):
        clean_link = None

        # If the link is a URL, check if it is within the same domain
        if re.search(HTTP_URL_PATTERN, link):
            # Parse the URL and check if the domain is the same
            url_obj = urlparse(link)
            if url_obj.netloc == local_domain:
                clean_link = link

        # If the link is not a URL, check if it is a relative link
        else:
            if link.startswith("/"):
                link = link[1:]
            elif link.startswith("#") or link.startswith("mailto:"):
                continue
            clean_link = "https://" + local_domain + "/" + link

        if clean_link is not None:
            if clean_link.endswith("/"):
                clean_link = clean_link[:-1]
            clean_links.append(clean_link)

    # Return the list of hyperlinks that are within the same domain
    return list(set(clean_links))
```

webcrawler.py

- Aimed at the openai.com domain, this script functions as a web crawler. It fetches web pages from the targeted domain and, with the assistance of the htmlparser.py script, extracts hyperlinks from these pages. The script then follows these hyperlinks, recursively crawling and collecting data from linked pages.

```
import requests
import os
from bs4 import BeautifulSoup
from collections import deque
from urllib.parse import urlparse
from htmlparser import get_domain_hyperlinks
import pandas as pd


domain = "openai.com"
full_url = "https://openai.com/"

def crawl(url):
    # Parse the URL and get the domain
    local_domain = urlparse(url).netloc

    # Create a queue to store the URLs to crawl
    queue = deque([url])

    # Create a set to store the URLs that have already been seen (no duplicates)
    seen = set([url])

    # Create a directory to store the text files
    if not os.path.exists("text/"):
            os.mkdir("text/")

    if not os.path.exists("text/"+local_domain+"/"):
            os.mkdir("text/" + local_domain + "/")

    # Create a directory to store the csv files
    if not os.path.exists("processed"):
            os.mkdir("processed")

    # While the queue is not empty, continue crawling
    while queue:

        # Get the next URL from the queue
        url = queue.pop()
        print(url) # for debugging and to see the progress

        # Save text from the url to a <url>.txt file
        with open('text/'+local_domain+'/'+url[8:].replace("/", "_") + ".txt", "w", encoding="UTF-8")
as f:

            # Get the text from the URL using BeautifulSoup
            soup = BeautifulSoup(requests.get(url).text, "html.parser")

            # Get the text but remove the tags
            text = soup.get_text()

            # If the crawler gets to a page that requires JavaScript, it will stop the crawl
            if ("You need to enable JavaScript to run this app." in text):
                print("Unable to parse page " + url + " due to JavaScript being required")

            # Otherwise, write the text to the file in the text directory
            f.write(text)

        # # Get the hyperlinks from the URL and add them to the queue
        for link in get_domain_hyperlinks(local_domain, url):
            if link not in seen:
                queue.append(link)
                seen.add(link)

crawl(full_url)


def remove_newlines(serie):
    serie = serie.str.replace('\n', ' ')
    serie = serie.str.replace('\\n', ' ')
    serie = serie.str.replace('  ', ' ')
    serie = serie.str.replace('  ', ' ')
    return serie

texts=[]

# Get all the text files in the text directory
for file in os.listdir("text/" + domain + "/"):

    # Open the file and read the text
    with open("text/" + domain + "/" + file, "r", encoding="UTF-8") as f:
        text = f.read()

        # Omit the first 11 lines and the last 4 lines, then replace -, _, and #update with spaces.
        texts.append((file[11:-4].replace('-',' ').replace('_', ' ').replace('#update',''), text))

# Create a dataframe from the list of texts
df = pd.DataFrame(texts, columns = ['fname', 'text'])

# Set the text column to be the raw text with the newlines removed
df['text'] = df.fname + ". " + remove_newlines(df.text)
df.to_csv('processed/scraped.csv')
df.head()
```

JavaScript and Web Application

- **index.js:** This is where the interaction happens on the client side. Users pose questions, which are then passed to the backend for processing. The returned answers are then displayed in a user-friendly manner.
- **generateAnswer.js:** the bridge between the frontend and backend. It's an API route that gets invoked when a user asks a question. This script interacts with the Python scripts and the OpenAI API to fetch the answers.

index.js

- Acting as the central frontend file, it offers an interface for user interaction within the web application. Users can input queries here, which are then sent to the backend through the generateAnswer.js API route. Once the backend processes these queries and fetches answers, the script ensures that these answers are displayed to the user in a structured manner.

```jsx
import Head from "next/head";
import { useState } from "react";
import styles from "./index.module.css";

export default function Home() {
  const [question, setQuestion] = useState("");
  const [result, setResult] = useState();
  const [loading, setLoading] = useState(false);

  async function onSubmit(event) {
    event.preventDefault();
    setLoading(true); // Start loading
    try {
      const response = await fetch("/api/generateAnswer", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({ question: question }),
      });

      const data = await response.json();
      if (response.status !== 200) {
        throw data.error || new Error(`Request failed with status ${response.status}`);
      }

      setResult(data.answer);
      setQuestion("");
    } catch(error) {
      console.error(error);
      alert(error.message);
    }
    setLoading(false); // End loading
  }

  return (
    <div>
      <Head>
        <title>Website Q&A</title>
        <link rel="icon" href="/qanda.png" />
      </Head>

      <main className={styles.main}>
        <img src="/qanda.png" className={styles.icon} />
        <h3>OpenAI Q&A Bot</h3>
        <form onSubmit={onSubmit}>
          <input
            type="text"
            name="question"
            placeholder="Enter a question"
            value={question}
            onChange={(e) => setQuestion(e.target.value)}
          />
          <input type="submit" value="Generate Answer" />
        </form>
        {loading && <div className={styles.spinner}></div>}
        <div className={styles.result}>{result}</div>
      </main>
    </div>
  );
}
```

generateAnswer.js

- This script functions as an intermediary between the frontend and backend. As an API route, it's designed to be invoked when a user submits a query. It takes this query, communicates with the Python scripts (particularly answer.py), and then interacts with the OpenAI API. The fetched answer is processed and sent back to the frontend for display.

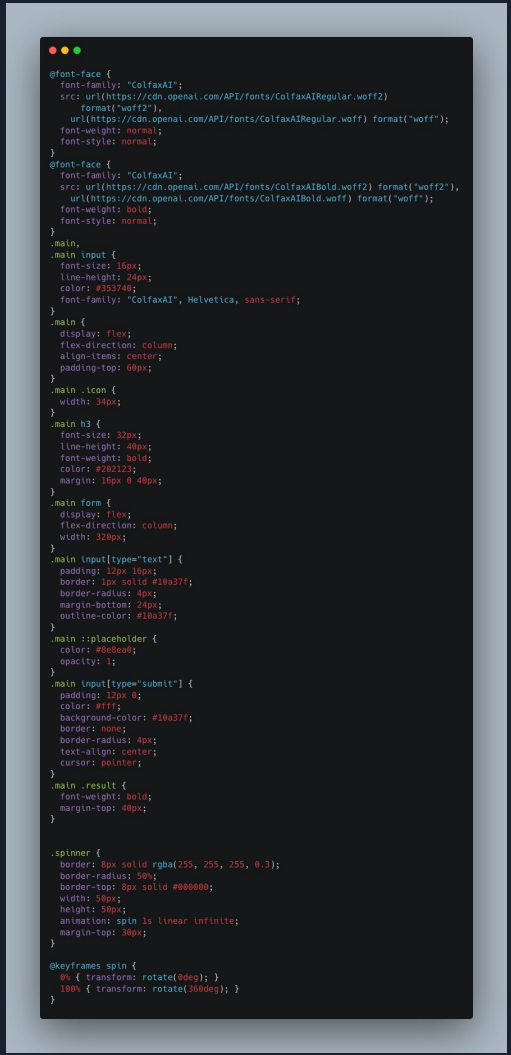
```
import { Configuration, OpenAIApi } from "openai";
import { exec } from 'child_process';


const configuration = new Configuration({
  apiKey: process.env.OPENAI_API_KEY,
});
const openai = new OpenAIApi(configuration);

export default async function (req, res) {
  if (!configuration.apiKey) {
    res.status(500).json({
      error: {
        message: "OpenAI API key not configured, please follow instructions in README.md",
      }
    });
    return;
  }

  const question = req.body.question || '';
  if (question.trim().length === 0) {
    res.status(400).json({
      error: {
        message: "Please enter a valid question",
      }
    });
    return;
  }

  const pythonProcess = exec(`python3 answer.py "${question}"`, (error, stdout, stderr) => {
    if (error) {
      console.error(`exec error ${error}`);
      res.status(500).json({ error: { message: "Internal Server Error" } });
      return;
    }

    if (stderr) {
      console.error(`Python script error: ${stderr}`);
  }

    const pythonResponse = stdout.trim();
    res.status(200).json({ answer: pythonResponse });

  });
}
```

Styling & Appearance

- **index.module.css**: This CSS module defines the look and feel of our application. From fonts to layouts, it ensures our application is visually coherent and user-friendly.

index.module.css

- This stylesheet module is dedicated to defining the visual elements of the web application. It contains various CSS rules and properties, ensuring that elements on the frontend are presented uniformly and appealingly, enhancing the overall user experience.



```css
@font-face {
  font-family: "ColfaxAI";
  src: url(https://cdn.openai.com/API/fonts/ColfaxAIRegular.woff2)
      format("woff2"),
    url(https://cdn.openai.com/API/fonts/ColfaxAIRegular.woff) format("woff");
  font-weight: normal;
  font-style: normal;
}
@font-face {
  font-family: "ColfaxAI";
  src: url(https://cdn.openai.com/API/fonts/ColfaxAIBold.woff2) format("woff2"),
    url(https://cdn.openai.com/API/fonts/ColfaxAIBold.woff) format("woff");
  font-weight: bold;
  font-style: normal;
}
.main,
.main input {
  font-size: 16px;
  line-height: 24px;
  color: #353740;
  font-family: "ColfaxAI", Helvetica, sans-serif;
}
.main {
  display: flex;
  flex-direction: column;
  align-items: center;
  padding-top: 60px;
}
.main .icon {
  width: 34px;
}
.main h3 {
  font-size: 32px;
  line-height: 40px;
  font-weight: bold;
  color: #202123;
  margin: 16px 0 40px;
}
.main form {
  display: flex;
  flex-direction: column;
  width: 320px;
}
.main input[type="text"] {
  padding: 12px 16px;
  border: 1px solid #10a37f;
  border-radius: 4px;
  margin-bottom: 24px;
  outline-color: #10a37f;
}
.main ::placeholder {
  color: #8e8ea0;
  opacity: 1;
}
.main input[type="submit"] {
  padding: 12px 0;
  color: #fff;
  background-color: #10a37f;
  border: none;
  border-radius: 4px;
  text-align: center;
  cursor: pointer;
}
.main .result {
  font-weight: bold;
  margin-top: 40px;
}


.spinner {
  border: 8px solid rgba(255, 255, 255, 0.3);
  border-radius: 50%;
  border-top: 8px solid #000000;
  width: 50px;
  height: 50px;
  animation: spin 1s linear infinite;
  margin-top: 30px;
}

@keyframes spin {
  0% { transform: rotate(0deg); }
  100% { transform: rotate(360deg); }
}
```

Configuration & Dependencies

- **package.json:** Node.js dependencies are listed here, ensuring the frontend runs seamlessly.
- **requirements.txt:** This file lists all the Python packages required for backend processing. From handling web requests to data manipulation, these packages are essential.

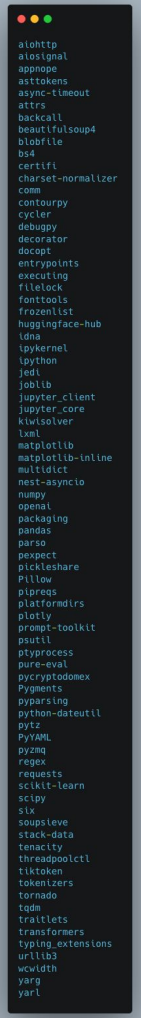
package.json

- This configuration file is essential for any Node.js project. It enumerates the dependencies that the frontend relies on. Beyond dependencies, the file also provides metadata about the project, scripts for execution, and other configuration details.

```
{
  "name": "openai-qa",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "crawl": "python3 webcrawler.py",
    "install":"pip install -r requirements.txt",
    "generateEmbed":"python3.py embeddings.py"
  },
  "dependencies": {
    "next": "^13.1.1",
    "openai": "^3.1.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  "engines": {
    "node": ">=16"
  }
}
```

requirements.txt

- This file is a list of Python packages and their respective versions that the backend scripts depend on. By having this file, it ensures that the backend, from API interactions to data processing, operates consistently across different setups.



Bibliography & Github Link

- https://github.com/Alami64/cs589_week2_hw2_nodejs/tree/master
- https://hc.labnet.sfbu.edu/~henry/sfbu/course/machine_learning/chatgpt/slide/exercise_chatgpt.html

Bibliography

- https://hc.labnet.sfbu.edu/~henry/sfbu/course/machine_learning/deep_learning/slide/exercise_deep_learning.html
- https://github.com/Alami64/Machine-Learning/tree/main/Deep%20Learning/Implementing%20Logic%20Gates%20using%20Neural%20Networks