Use ChatGPT to create customer support website

Submitted by Manish Bafna
Student Id: 19655
Instructor: Dr. Henry Chang
GIT: ChatGPT

Table of Content

Introduction

This tutorial walks through a simple example of crawling a website (in this example, the OpenAI website), turning the crawled pages into embeddings using the Embeddings API, and then creating a basic search functionality that allows a user to ask questions about the embedded information. This is intended to be a starting point for more sophisticated applications that make use of custom knowledge bases.

What are embeddings?

OpenAI's text embeddings measure the relatedness of text strings. Embeddings are commonly used for:

Search (where results are ranked by relevance to a query string)

Clustering (where text strings are grouped by similarity)

Introduction

Recommendations (where items with related text strings are recommended)

Anomaly detection (where outliers with little relatedness are identified)

Diversity measurement (where similarity distributions are analyzed)

Classification (where text strings are classified by their most similar label)

An embedding is a vector (list) of floating point numbers. The distance between two vectors measures their relatedness. Small distances suggest high relatedness and large distances suggest low relatedness.

Design

This code is a web crawler that extracts text data from web pages within a given domain and saves the text to a file. The crawler starts with a root URL (specified by full_url) and finds all hyperlinks on that page that are within the same domain. It then visits each of those hyperlinks and repeats the process, effectively crawling the entire domain. The crawler saves the text from each page to a file in the text/ directory, using the URL path as the file name.

The second part of the code reads the text files from the text/ directory, removes unnecessary characters, and saves the cleaned text to a pandas DataFrame. The cleaned text is used for further analysis.

Design

Here is a brief overview of the different parts of the code:

Imports necessary libraries including requests, re, urllib.request, BeautifulSoup, deque, HTMLParser, urlparse, os and time

Defines a regex pattern to match URLs and specifies the root domain to crawl.

Creates a class called HyperlinkParser that extends HTMLParser and overrides the handle_starttag method to get hyperlinks.

Defines a function called get_hyperlinks that uses urllib.request to open a URL, reads the HTML, and returns a list of hyperlinks using the HyperlinkParser.

Design

Defines a function called get_domain_hyperlinks that takes a local domain and a URL, and returns a list of hyperlinks that are within the same domain.

Defines a function called crawl that takes a URL, parses the domain, creates a queue to store URLs to crawl, and a set to store URLs that have already been seen, and crawls through the URLs until the queue is empty. It saves the text from each URL to a text file using BeautifulSoup, and gets the hyperlinks from the URL and adds them to the queue.

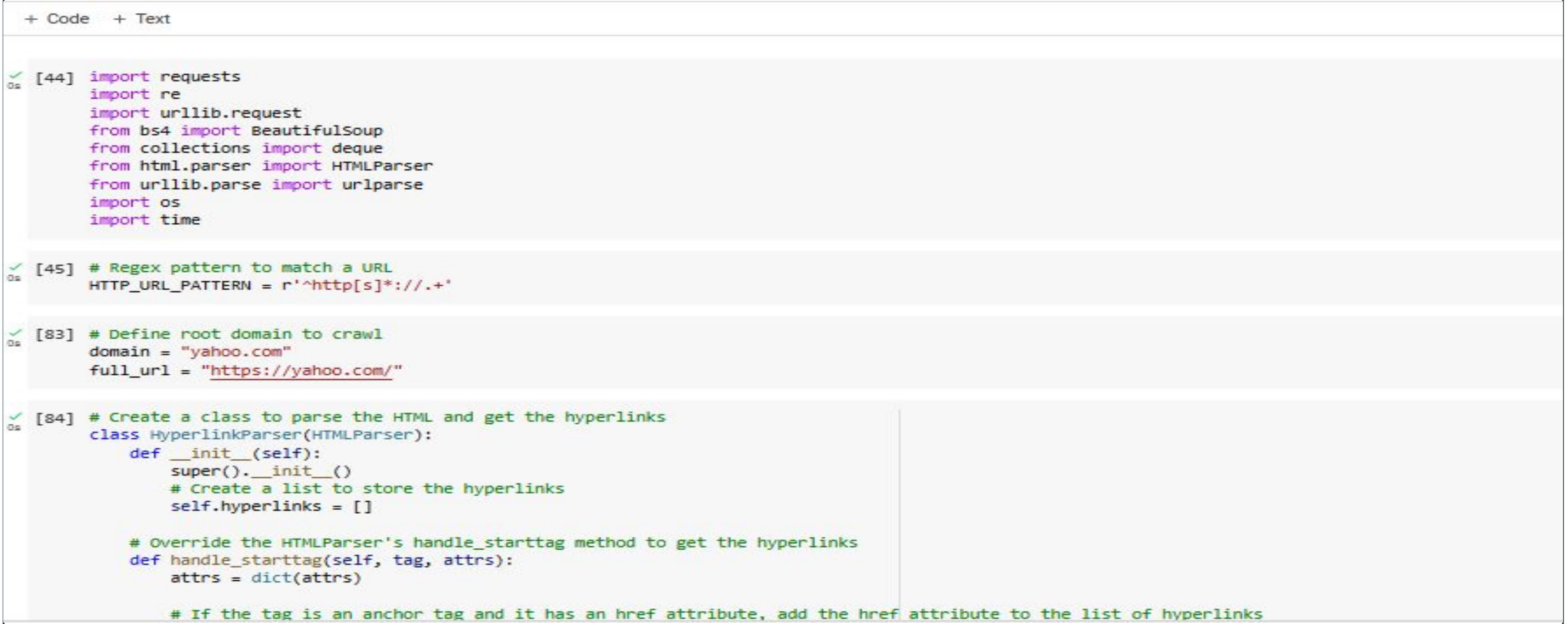
Defines a function called remove_newlines that replaces newline characters with spaces in a pandas series.

The script uses pandas to read the text files and creates a list of tuples with the filename and the cleaned text.

The cleaned text is saved to a CSV file.

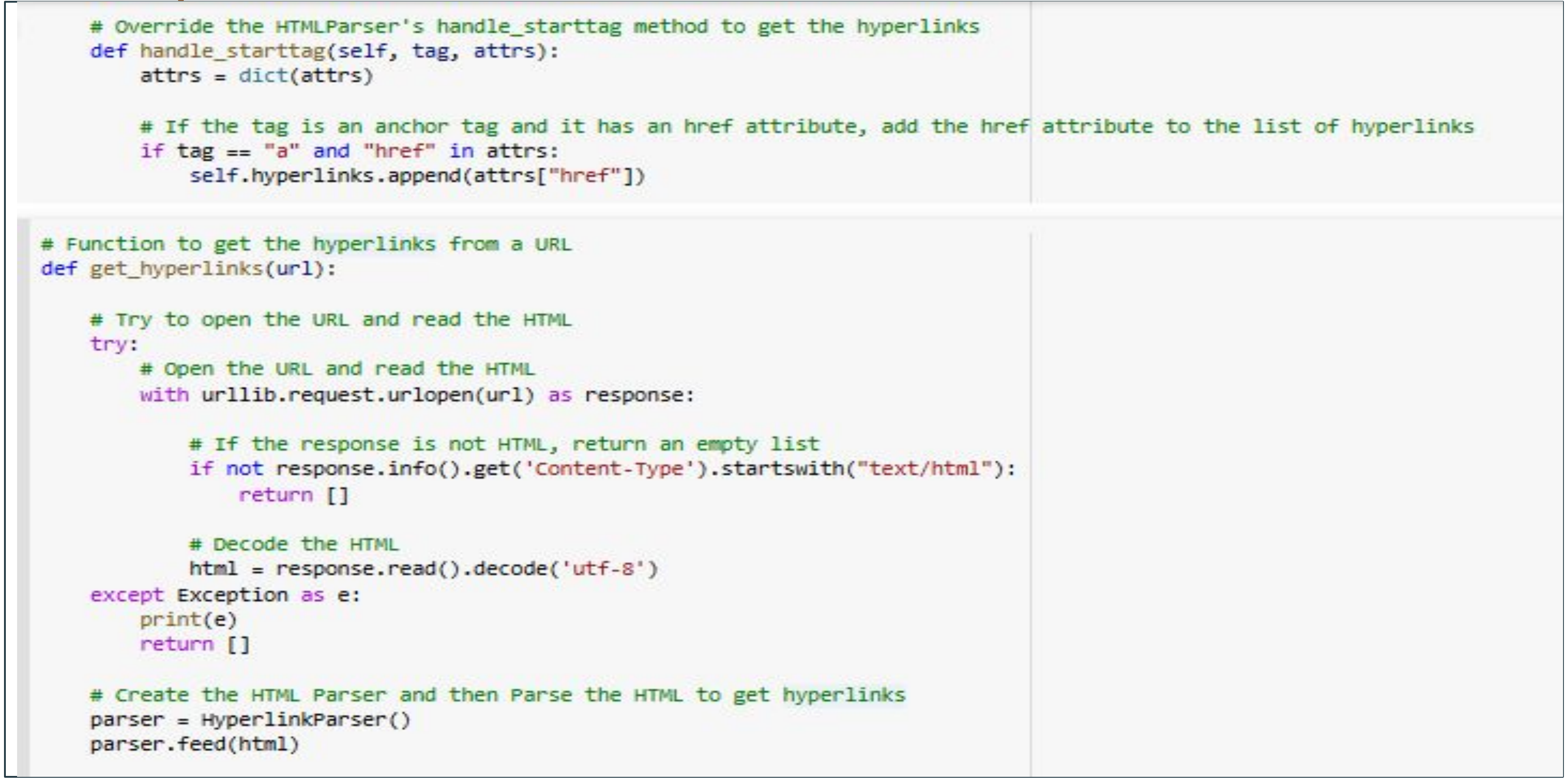
Implementation

We use google colab to generate a python code and use the yahoo.com website as our test website



```
[44] import requests
     import re
     import urllib.request
     from bs4 import BeautifulSoup
     from collections import deque
     from html.parser import HTMLParser
     from urllib.parse import urlparse
     import os
     import time

[45] # Regex pattern to match a URL
     HTTP_URL_PATTERN = r'^http[s]*://.+'

[83] # Define root domain to crawl
     domain = "yahoo.com"
     full_url = "https://yahoo.com/"

[84] # Create a class to parse the HTML and get the hyperlinks
     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, attrs):
             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
```

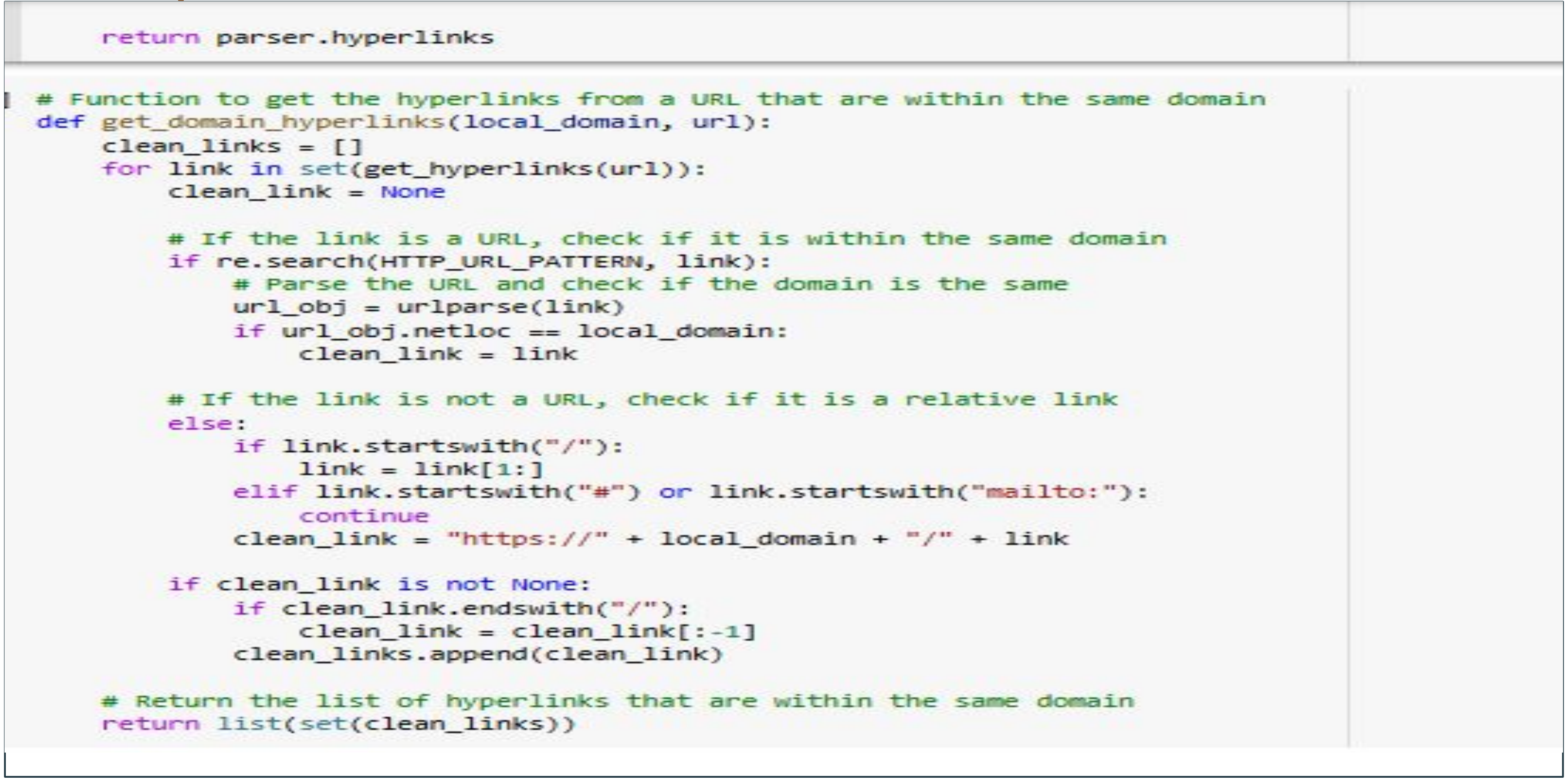
Implementation



```
    # Override the HTMLParser's handle_starttag method to get the hyperlinks
    def handle_starttag(self, tag, attrs):
        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)
```

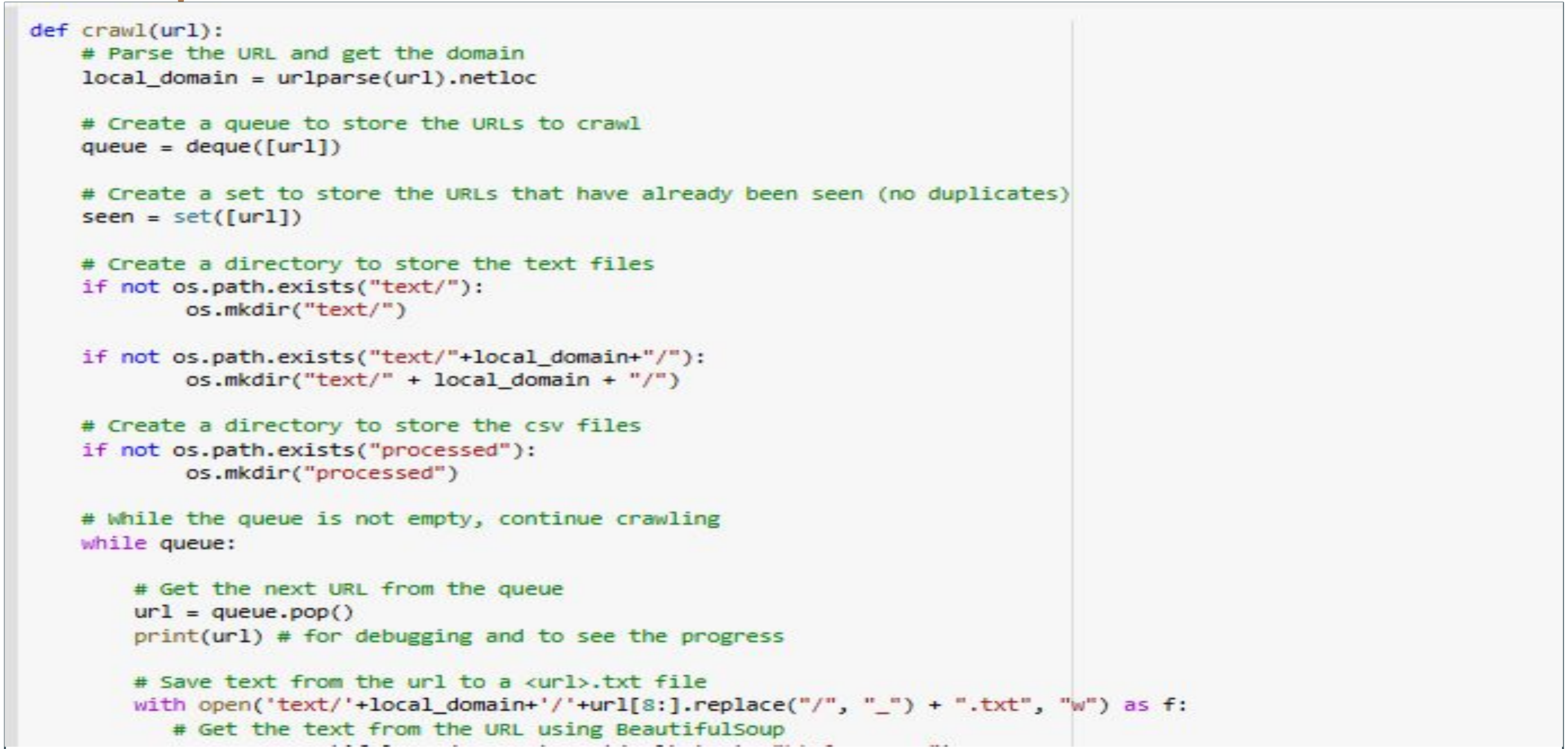
Implementation



```
    return parser.hyperlinks
```

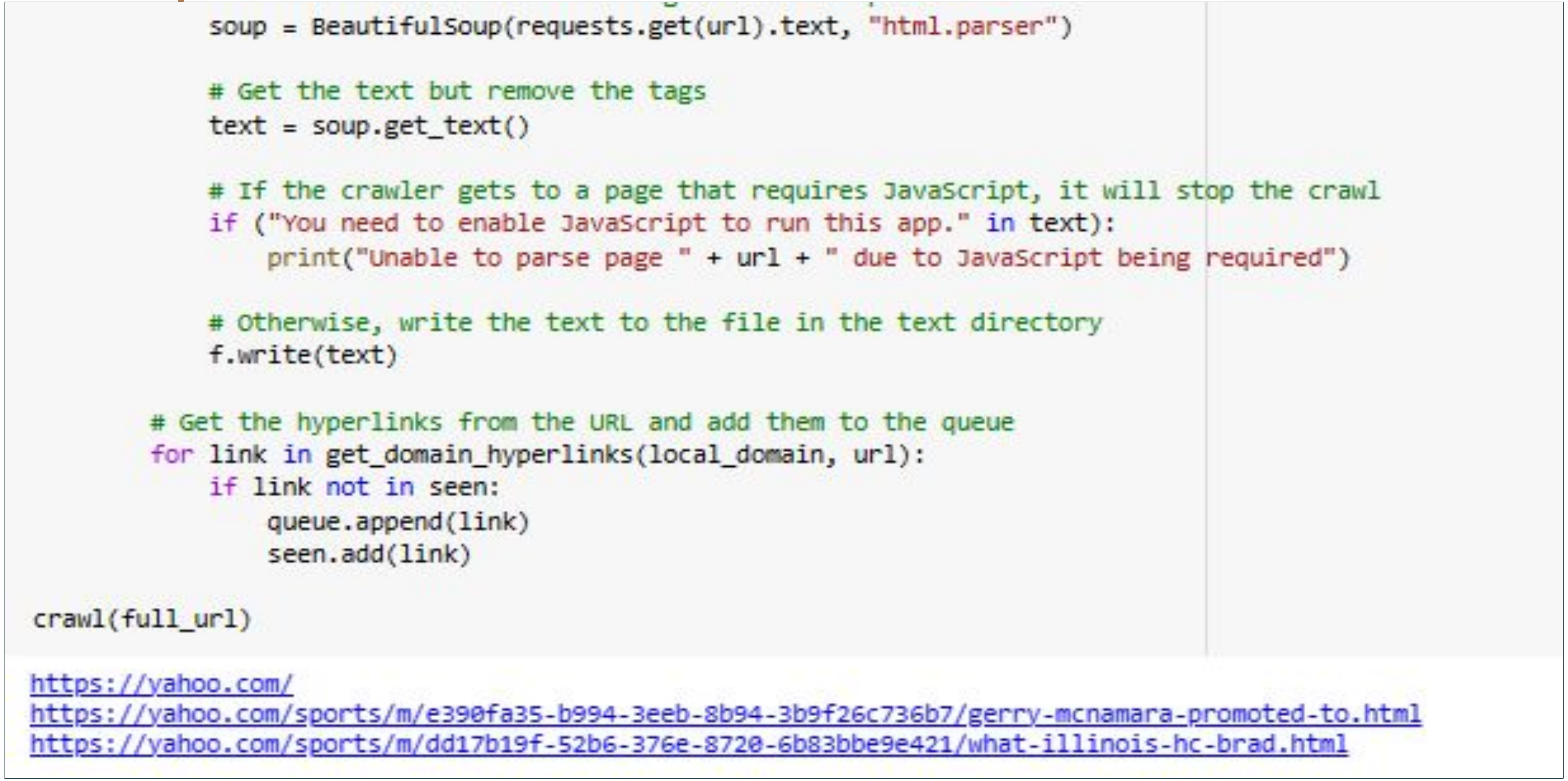
```
# Function to get the hyperlinks from a URL that are within the same domain
def get_domain_hyperlinks(local_domain, url):
    clean_links = []
    for link in set(get_hyperlinks(url)):
        clean_link = None

        # 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))
```

Implementation



```
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") as f:
            # Get the text from the URL using BeautifulSoup
```

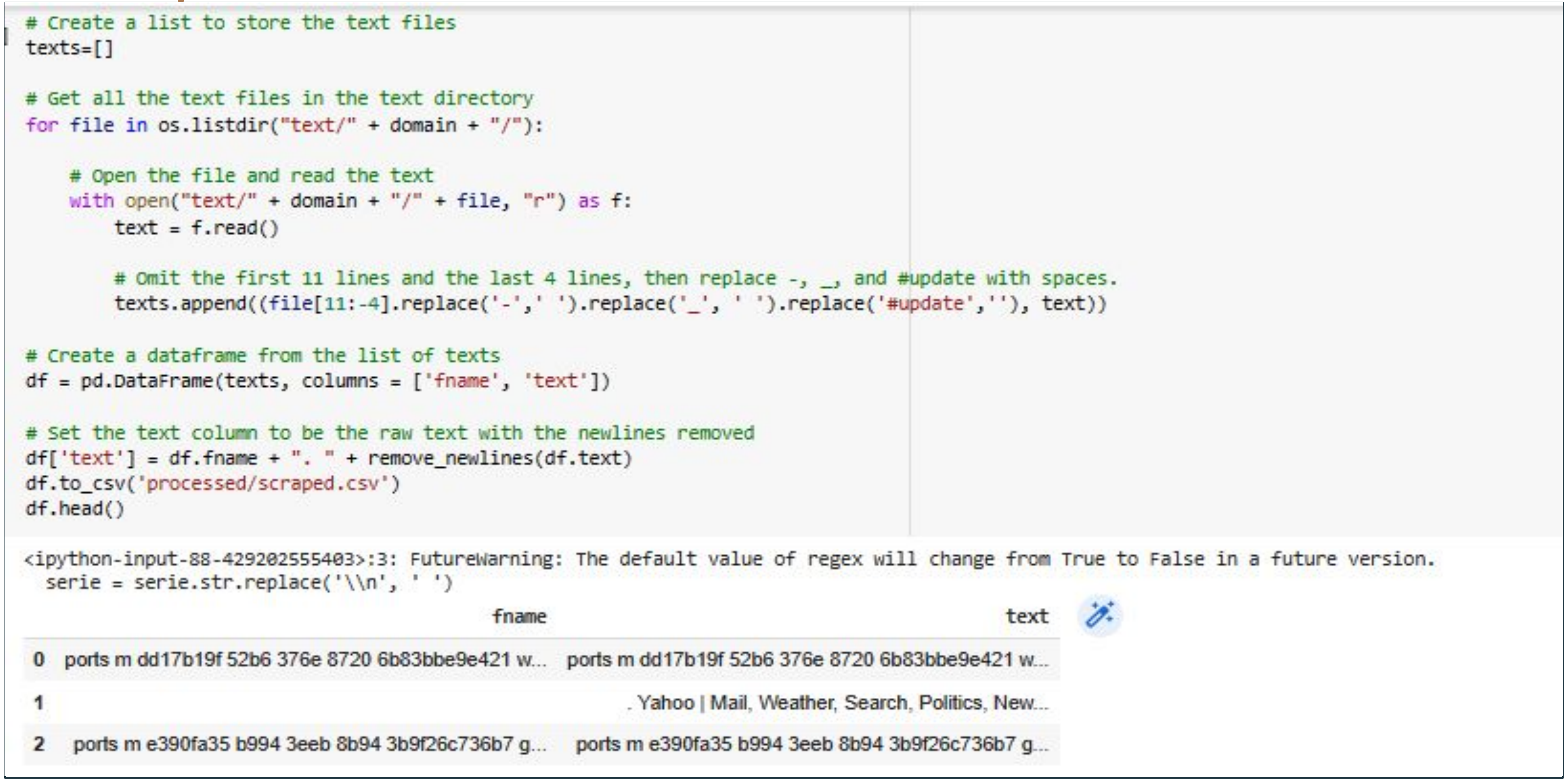
Implementation



```
            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)
```

```
https://yahoo.com/
https://yahoo.com/sports/m/e390fa35-b994-3eeb-8b94-3b9f26c736b7/gerry-mcnamara-promoted-to.html
https://yahoo.com/sports/m/dd17b19f-52b6-376e-8720-6b83bbe9e421/what-illinois-hc-brad.html
```

Implementation

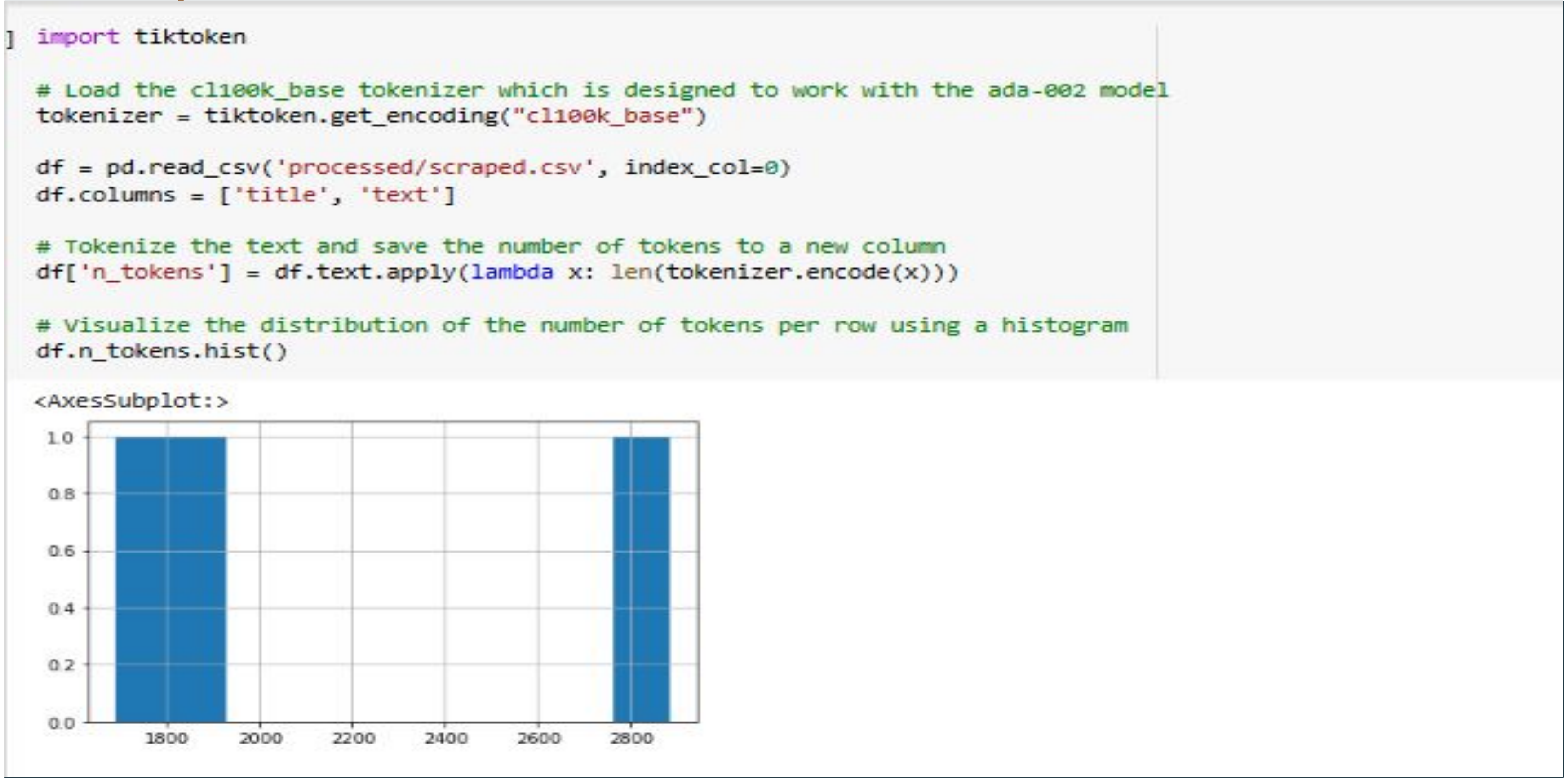


```python
# Create a list to store the text files
texts=[]

# 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") 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()
```

```
<ipython-input-88-429202555403>:3: FutureWarning: The default value of regex will change from True to False in a future version.
  serie = serie.str.replace('\\n', ' ')
```

	fname	text
0	ports m dd17b19f 52b6 376e 8720 6b83bbe9e421 w...	ports m dd17b19f 52b6 376e 8720 6b83bbe9e421 w...
1		. Yahoo | Mail, Weather, Search, Politics, New...
2	ports m e390fa35 b994 3eeb 8b94 3b9f26c736b7 g...	ports m e390fa35 b994 3eeb 8b94 3b9f26c736b7 g...

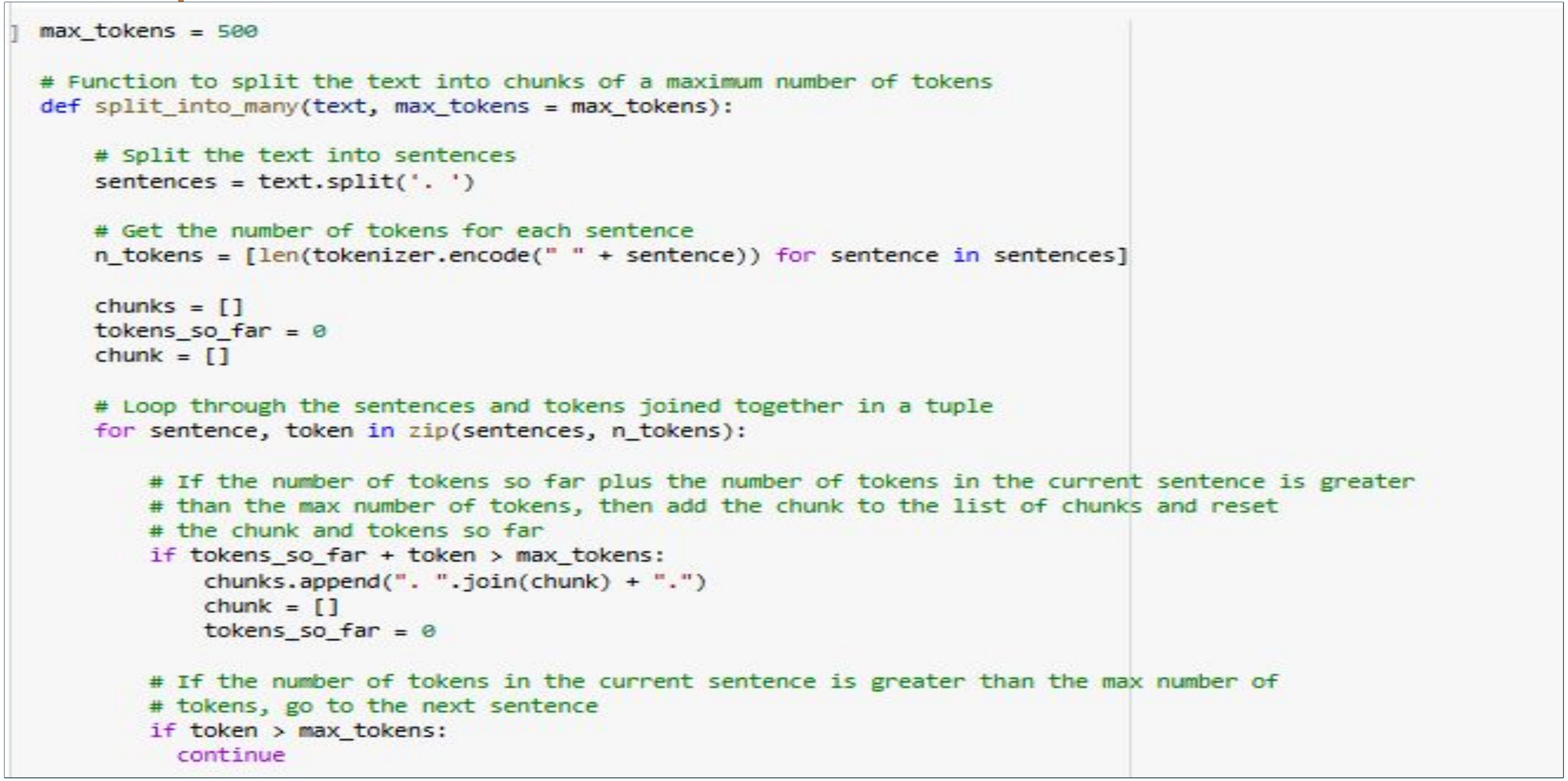
Implementation



```
]  import tiktoken

   # 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)))

   # Visualize the distribution of the number of tokens per row using a histogram
   df.n_tokens.hist()
```

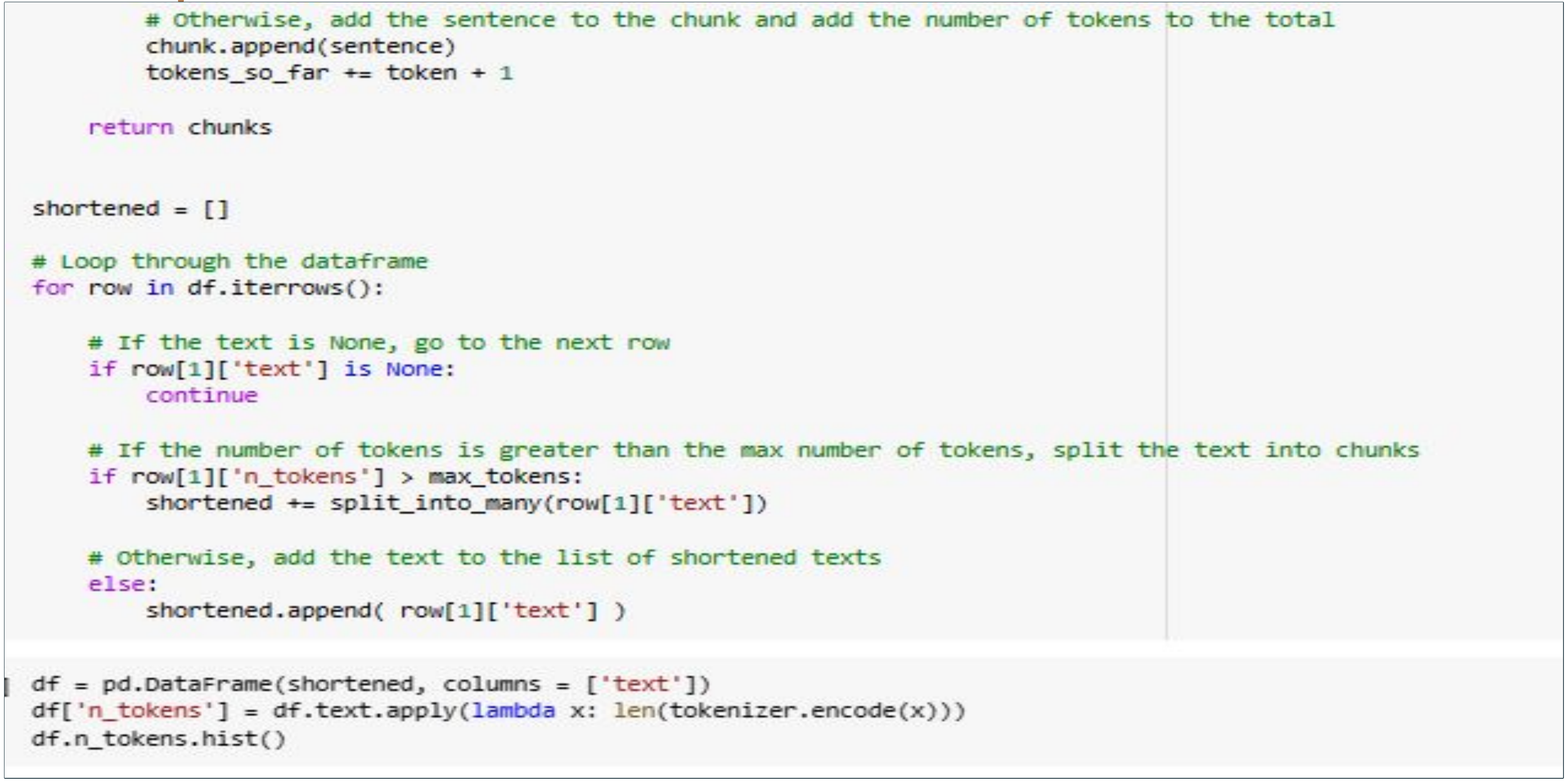
```
<AxesSubplot:>
```

Implementation



```
] 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
```

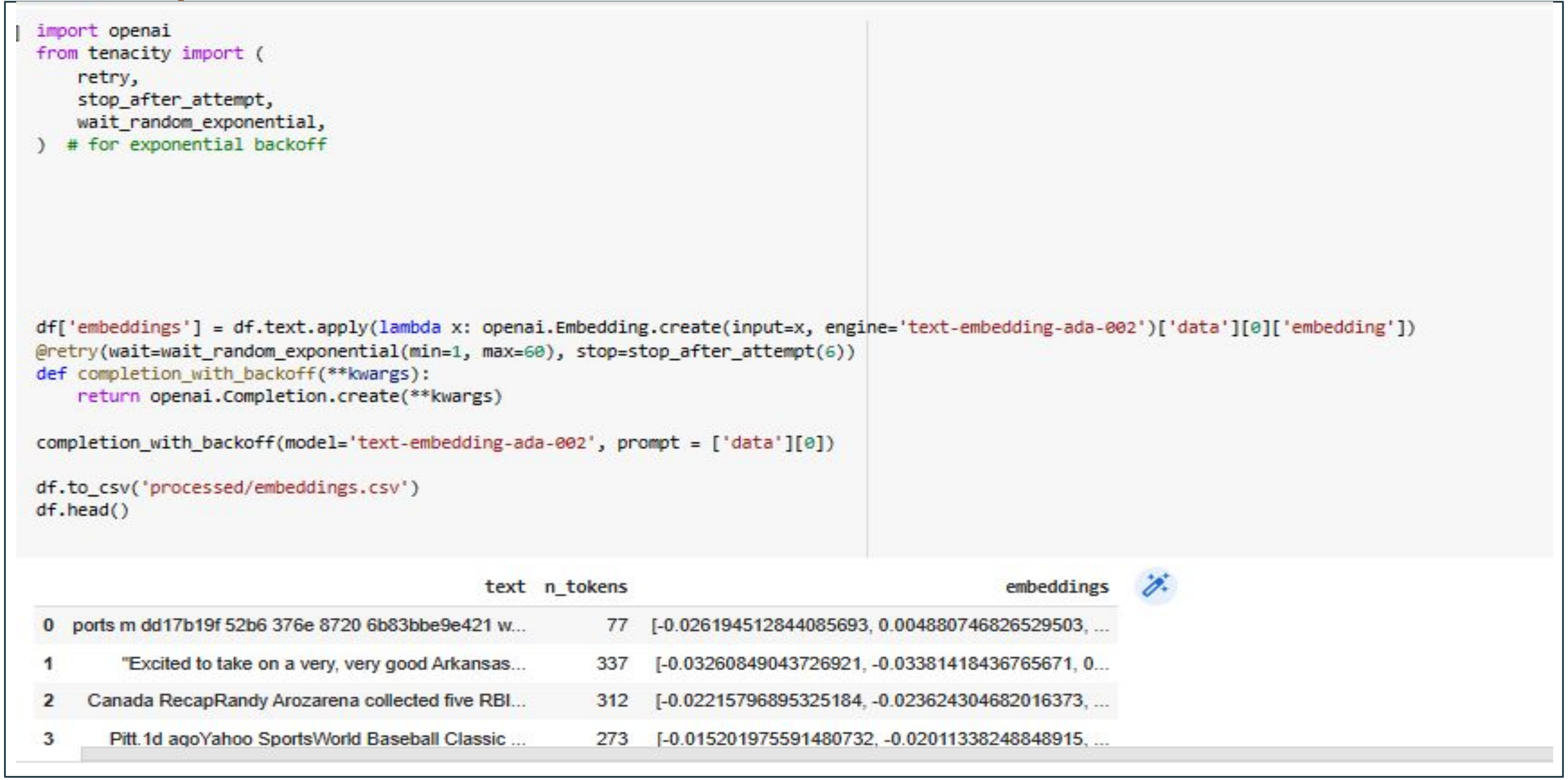
Implementation



```
        # 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)))
df.n_tokens.hist()
```

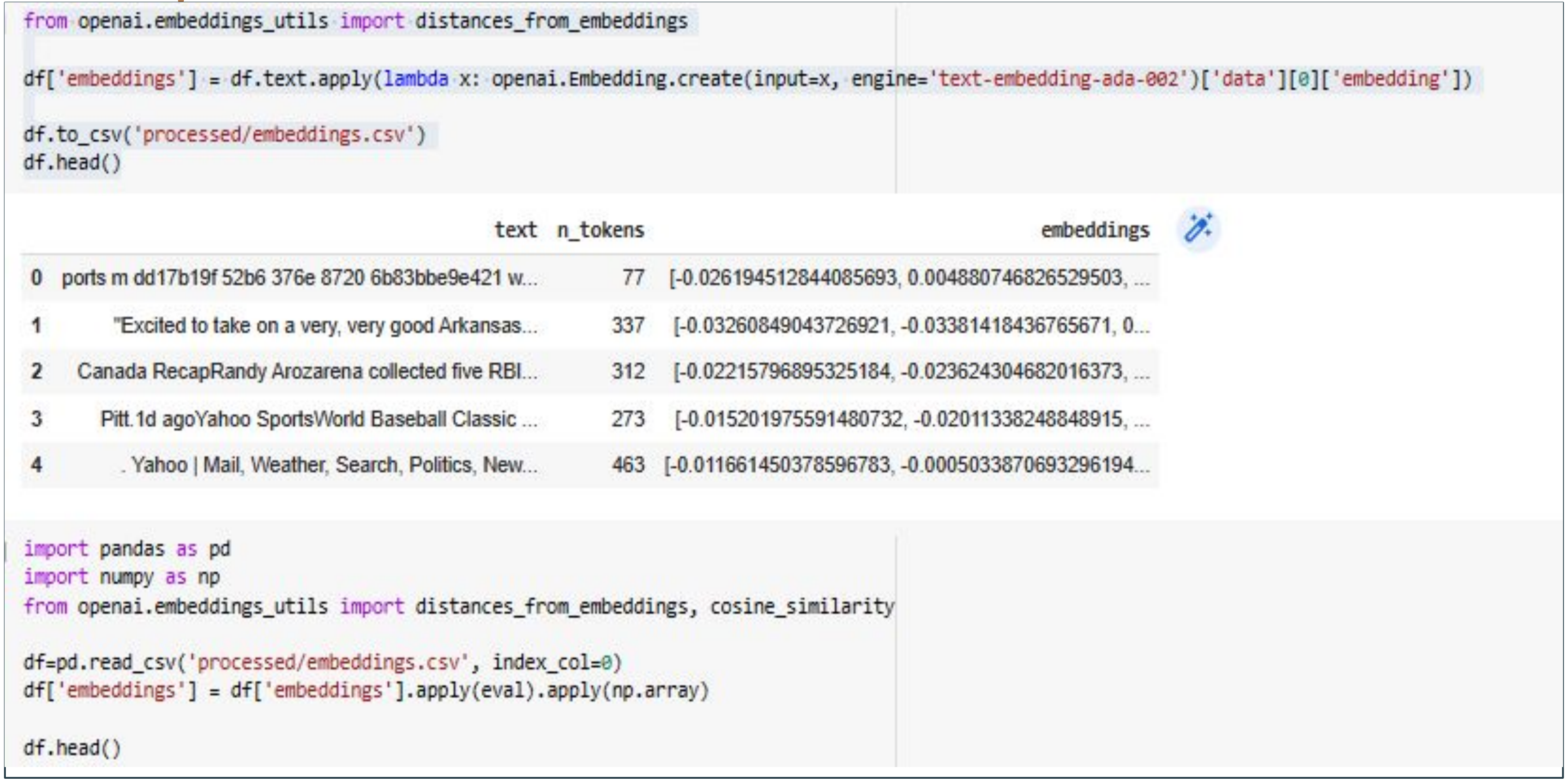
Implementation



```python
import openai
from tenacity import (
    retry,
    stop_after_attempt,
    wait_random_exponential,
)  # for exponential backoff







df['embeddings'] = df.text.apply(lambda x: openai.Embedding.create(input=x, engine='text-embedding-ada-002')['data'][0]['embedding'])
@retry(wait=wait_random_exponential(min=1, max=60), stop=stop_after_attempt(6))
def completion_with_backoff(**kwargs):
    return openai.Completion.create(**kwargs)

completion_with_backoff(model='text-embedding-ada-002', prompt = ['data'][0])

df.to_csv('processed/embeddings.csv')
df.head()
```

	text	n_tokens	embeddings
0	ports m dd17b19f 52b6 376e 8720 6b83bbe9e421 w...	77	[-0.026194512844085693, 0.004880746826529503, ...
1	"Excited to take on a very, very good Arkansas...	337	[-0.03260849043726921, -0.03381418436765671, 0...
2	Canada RecapRandy Arozarena collected five RBI...	312	[-0.02215796895325184, -0.023624304682016373, ...
3	Pitt.1d agoYahoo SportsWorld Baseball Classic ...	273	[-0.015201975591480732, -0.02011338248848915, ...

Implementation

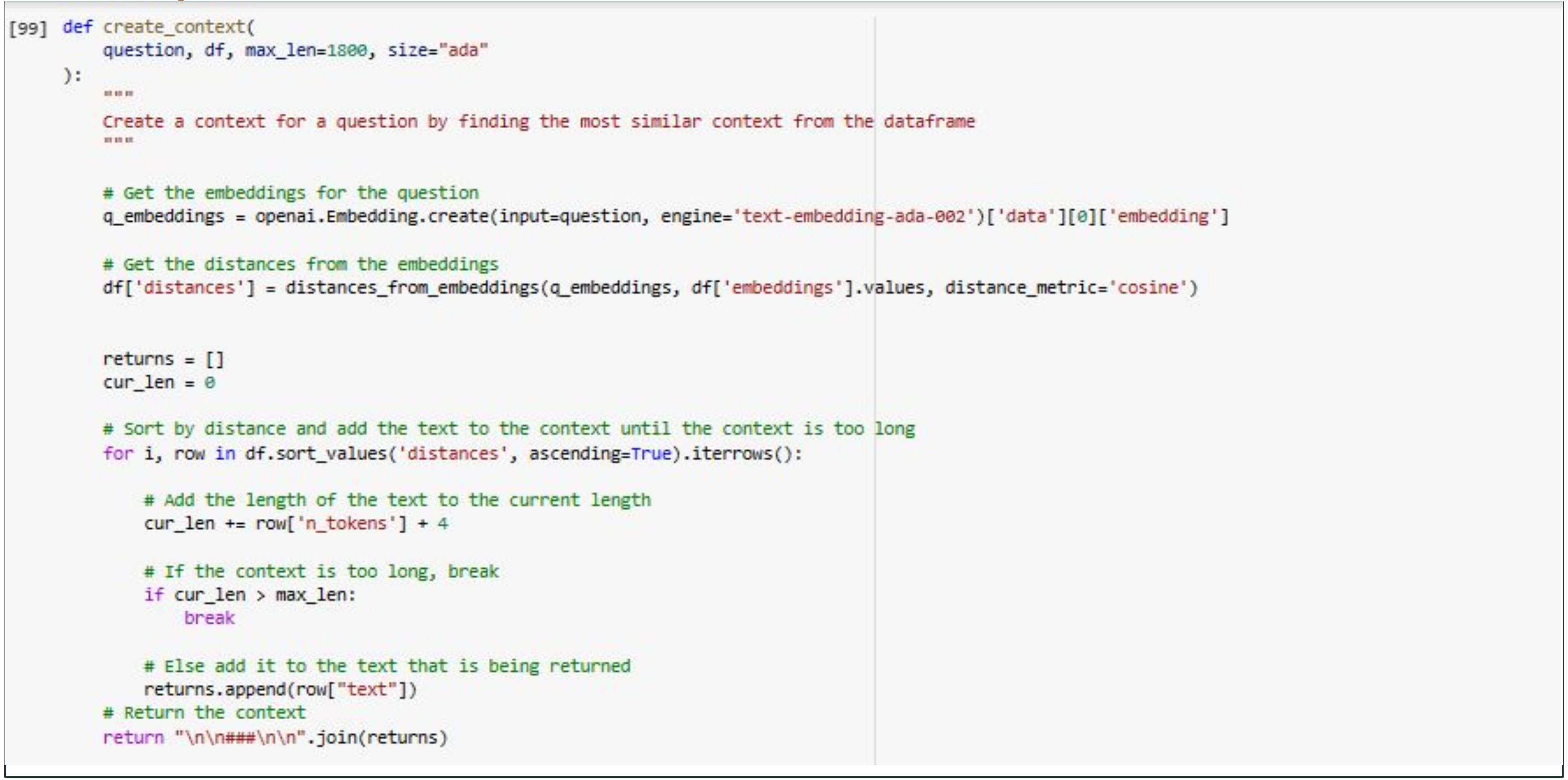


```
from openai.embeddings_utils import distances_from_embeddings

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')
df.head()
```

```
                                              text  n_tokens                                        embeddings
0  ports m dd17b19f 52b6 376e 8720 6b83bbe9e421 w...        77  [-0.026194512844085693, 0.004880746826529503, ...
1          "Excited to take on a very, very good Arkansas...       337  [-0.03260849043726921, -0.03381418436765671, 0...
2    Canada RecapRandy Arozarena collected five RBI...       312  [-0.02215796895325184, -0.02362430468201637З, ...
3       Pitt.1d agoYahoo SportsWorld Baseball Classic ...       273  [-0.015201975591480732, -0.02011338248848915, ...
4            . Yahoo | Mail, Weather, Search, Politics, New...       463  [-0.011661450378596783, -0.0005033870693296194...
```

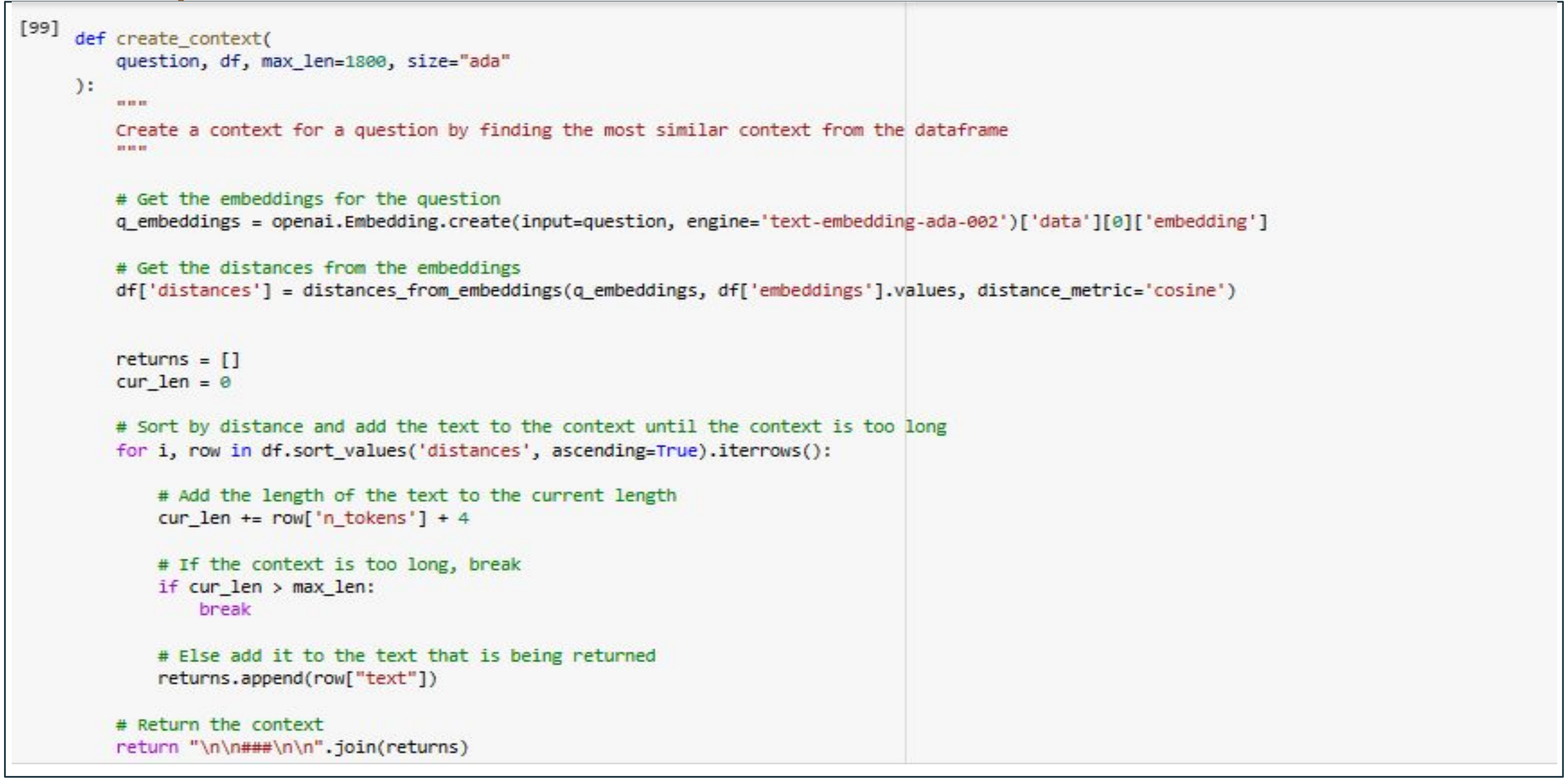
```
import pandas as pd
import numpy as np
from openai.embeddings_utils import distances_from_embeddings, cosine_similarity

df=pd.read_csv('processed/embeddings.csv', index_col=0)
df['embeddings'] = df['embeddings'].apply(eval).apply(np.array)

df.head()
```

Implementation



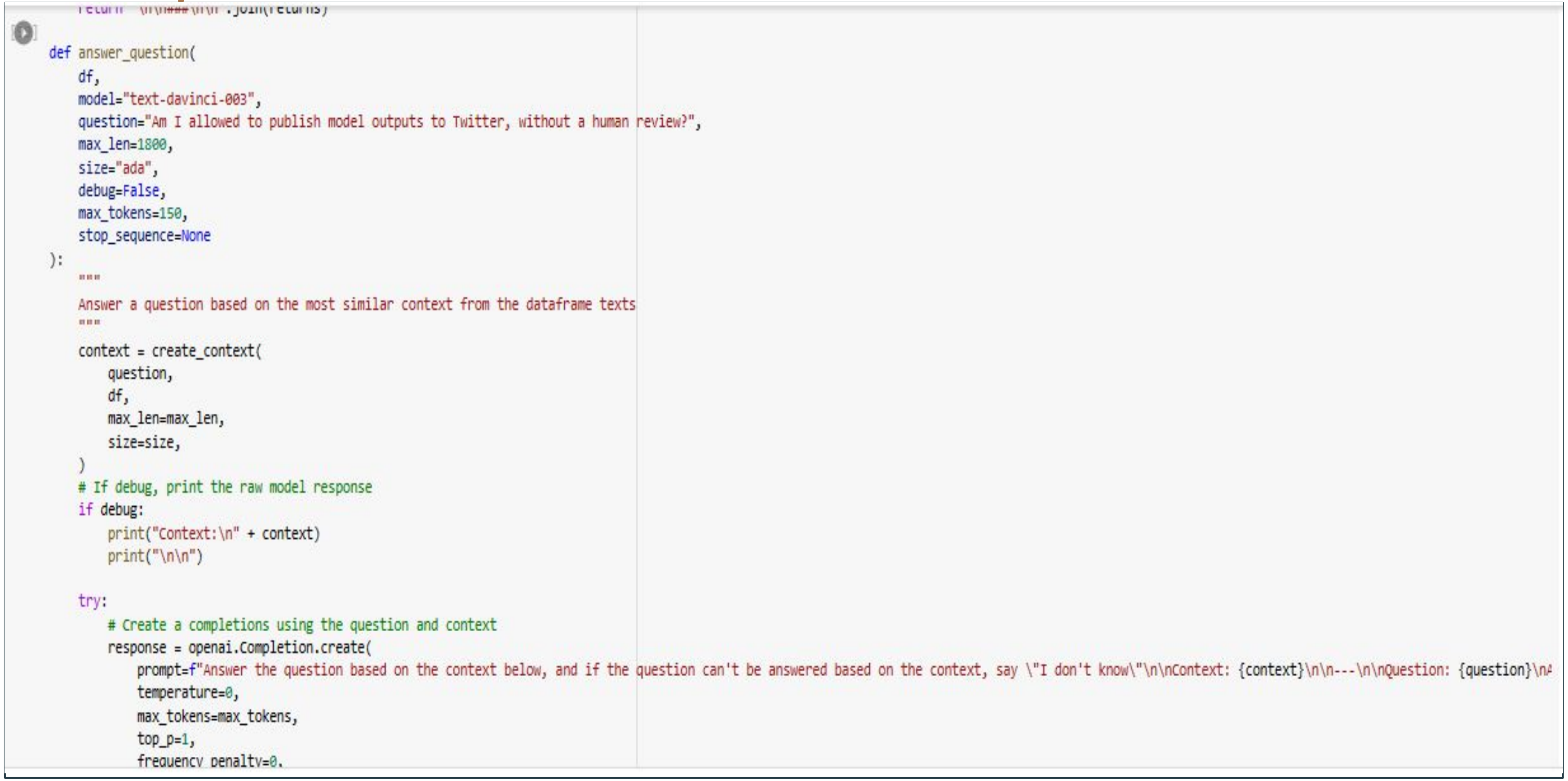
```
[99] def create_context(
         question, df, max_len=1800, size="ada"
     ):
         """
         Create a context for a question by finding the most similar context from the dataframe
         """

         # 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)
```

Implementation



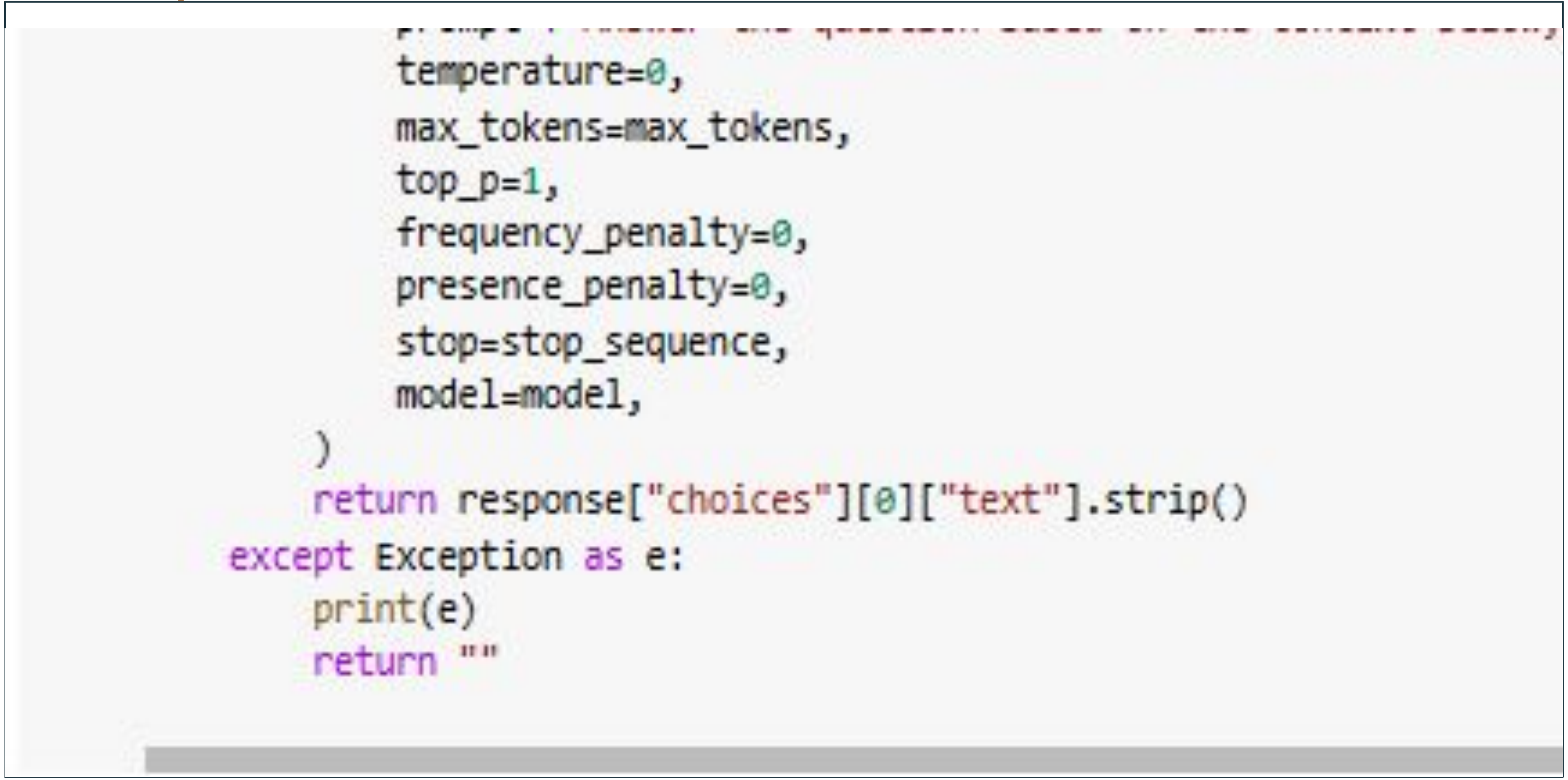
```
[99] def create_context(
         question, df, max_len=1800, size="ada"
     ):
         """
         Create a context for a question by finding the most similar context from the dataframe
         """

         # 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)
```

Implementation



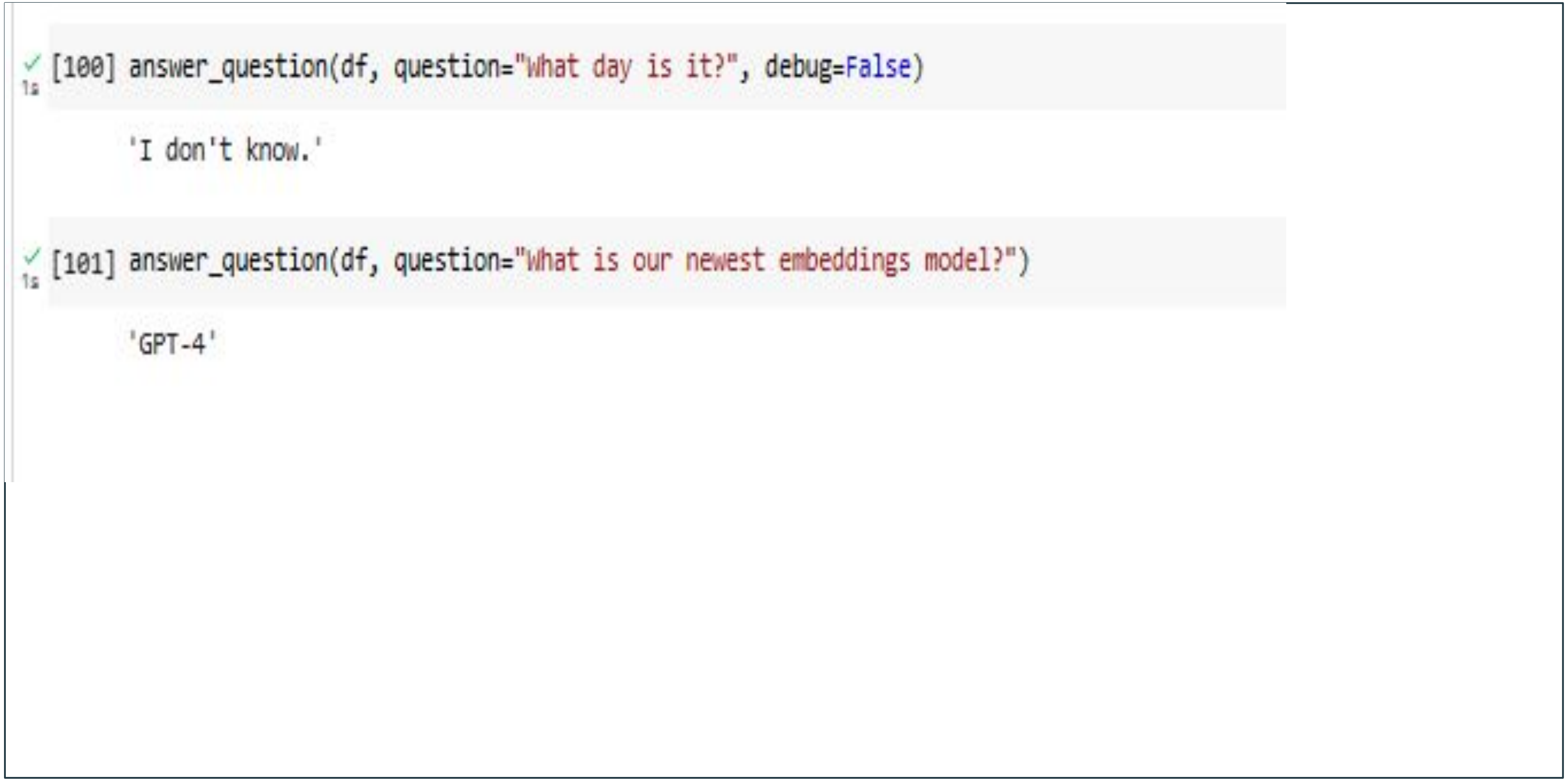
```python
    return "\n\n###\n\n".join(returns)

def answer_question(
    df,
    model="text-davinci-003",
    question="Am I allowed to publish model outputs to Twitter, without a human review?",
    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.Completion.create(
            prompt=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\"\n\nContext: {context}\n\n---\n\nQuestion: {question}\nA
            temperature=0,
            max_tokens=max_tokens,
            top_p=1,
            frequency_penalty=0,
```

Implementation



```
            temperature=0,
            max_tokens=max_tokens,
            top_p=1,
            frequency_penalty=0,
            presence_penalty=0,
            stop=stop_sequence,
            model=model,
        )
        return response["choices"][0]["text"].strip()
    except Exception as e:
        print(e)
        return ""
```

Test



```
[100] answer_question(df, question="What day is it?", debug=False)

'I don't know.'

[101] answer_question(df, question="What is our newest embeddings model?")

'GPT-4'
```

Enhancement

- Crawl more efficiently: The current implementation of the crawler is not very efficient. It visits every page on the domain, even if it has already been visited. To make the crawler more efficient, you could keep track of which pages have already been visited and skip them on subsequent crawls. You could also prioritize pages based on their relevance to the domain, such as by using a page ranking algorithm.
- Store data in a database: Currently, the crawler stores text files on disk. To make it easier to analyze the data, you could store it in a database instead. This would allow you to query the data more easily and perform more complex analyses.Store data in a database: Currently, the crawler stores text files on disk. To make it easier to analyze the data, you could store it in a database instead. This would allow you to query the data more easily and perform more complex analyses.

Enhancement

- Use a more advanced parser: The current implementation of the parser uses a simple regular expression to match URLs. A more advanced parser, such as one based on a state machine, could be more reliable and accurate.
- Add support for different file formats: Currently, the crawler only supports HTML pages. To make it more versatile, you could add support for other file formats, such as PDF or Microsoft Word documents.
- Add support for different types of analysis: Currently, the code only performs basic text cleaning and analysis. You could expand it to support more advanced analyses, such as sentiment analysis or topic modeling.

Conclusion

Firstly, we need to define the scope of the AI's capabilities, such as the types of questions it can answer and the information it needs to access on our website. Next, we need to gather and structure the data that the AI will use to answer questions. This may involve creating a knowledge base or database of information about our website and its content.

Then, we need to select or develop an appropriate AI technology, such as a chatbot or natural language processing system, and train it on our data set. This may involve using machine learning algorithms to improve the AI's accuracy and effectiveness.

Finally, we need to integrate the AI into our website and make it accessible to users, such as by adding a chatbot interface or a search function that uses natural language processing. We may also need to monitor and adjust the AI's performance over time to ensure that it continues to provide accurate and useful answers to user questions.

Overall, building an AI that can answer questions about our website requires careful planning, data preparation, technology selection and training, and ongoing monitoring and improvement.

References

Web Q&A - OpenAI API

openai-cookbook/How_to_handle_rate_limits.ipynb at main · openai/openai-cookbook · GitHub