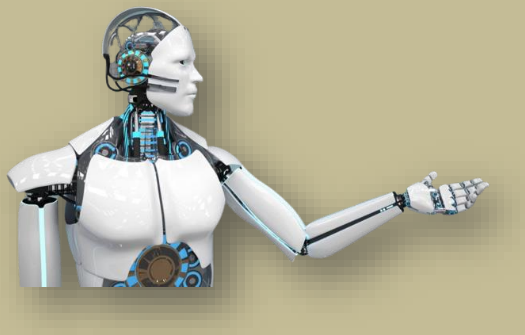


Converting PDFs data into OpenAI embeddings using LangChain

Nisar Ahmad

Table of Contents

Converting PDFs data into OpenAI embeddings using LangChain 0

Introduction: 2

System Requirements: 3

Hardware Requirements: 3

Software Requirements: 3

Installation and Setup: 4

LangChain Setup: 4

1. **Installation** 4
2. **Configuration** 4

OpenAI API Setup: 4

1. **Obtain API Key** 5
2. **Install OpenAI Python Client** 5

PDF Data Extraction: 5

Code Overview: 5

Import packages: 5

1. **os** 6
2. **OpenAIEmbeddings** 6
3. **CharacterTextSplitter** 6
4. **ElasticVectorSearch** 6
5. **load_qa_chain** 6
6. **OpenAI** 6
7. **re** 6
8. **nltk** 6
9. **PyPDF2** 7

Extracting, Cleaning, and Segmenting Text from PDF Files: 7

Building Section Dictionary and Text Embedding: 11

Building Section Dictionary: 12

Preparing Texts for Embedding: 12

Setting up OpenAI API: 12

Initializing Vector Store: 13

Loading Question Answering Chain: 13

Asking Questions: 13

Conclusion: 13

Introduction:

This document provides an in-depth analysis and explanation of the codebase designed to convert PDF data into OpenAI embeddings using LangChain. The overarching objective of this project is to leverage artificial intelligence and machine learning techniques to transform unstructured data from PDFs into more structured and analyzable data, specifically OpenAI embeddings.

In the era of big data, having access to structured data is pivotal. It enables sophisticated and meaningful analyses, which in turn drive informed decision-making. This is where our project steps in. By converting PDF data into OpenAI embeddings, we can provide a more structured representation of the data, making it easier to work with for various applications such as information retrieval, sentiment analysis, document summarization, and more.

The tool of our choice for this task is LangChain, a cutting-edge technology that facilitates a streamlined conversion process. LangChain is a scalable and robust solution that uses advanced machine learning algorithms and natural language processing techniques to convert and structure data efficiently.

OpenAI embeddings, the output of our process, are high-dimensional vectors that represent the semantic content of the PDF data. These embeddings are generated by a neural network model trained on vast amounts of text data, making it possible to capture complex patterns and relationships in the data that traditional approaches might miss.

The rest of this documentation will walk you through the design, implementation, and operation of our codebase. We will delve into the specifics of how we extract data from PDFs, how we use LangChain to convert this data, and how we generate and use OpenAI embeddings. By the end of this documentation, you will have a comprehensive understanding of our codebase and the underlying principles that drive it.

System Requirements:

Hardware Requirements:

Running this project efficiently requires a robust hardware setup. The specific requirements will depend on the size of the PDF data to be processed, but below are the general hardware requirements:

Processor: A modern multi-core processor (Intel i5, i7, or Xeon, or AMD equivalent). This aids in the parallel processing of data, significantly reducing the time required for data conversion.

RAM: A minimum of 8GB RAM is recommended. If you're working with large PDF files or numerous files simultaneously, 16GB or more might be required. The RAM is crucial for storing temporary data during processing.

Storage: Sufficient hard drive space (SSD preferred for faster read/write operations) is needed for storing the PDF files, intermediate data, and the final embeddings. The specific amount will depend on the volume of your data. We can also use any cloud platform to store the embeddings, for example Pinecone.

GPU: While not strictly necessary, a dedicated GPU (such as NVIDIA Tesla, Titan, or Quadro series) can significantly accelerate machine learning computations, especially for the generation of OpenAI embeddings. You can use Google Colab for this and use their free GPU.

Software Requirements:

The software requirements for this project are as follows:

Operating System: This project is platform-independent and can be run on any modern operating system such as Windows, MacOS, or Linux.

Python: As the primary programming language used in this project, Python 3.8 or later is required.

Python Libraries: Several Python libraries are essential for running this project, including but not limited to:

- **PyPDF2 or PDFMiner**: For reading and extracting data from PDF files.
- **LangChain**: For converting the extracted text data.
- **openai**: For generating embeddings using OpenAI's GPT-4 model.
- **numpy**, pandas: For data manipulation and analysis.

- **scikit-learn**: For any additional machine learning tasks on the embeddings.

OpenAI API Key: To use OpenAI's GPT-4 model for generating embeddings, you need an API key from OpenAI. This key is usually provided as part of OpenAI's subscription services.

IDE/Text Editor: Any text editor or Integrated Development Environment (IDE) that supports Python will be needed to view and edit the code. Examples include Jupyter Notebook, PyCharm, Visual Studio Code, and Sublime Text.

Version Control System: Although not mandatory, a version control system like Git is highly recommended for managing different versions of the codebase.

Please ensure that all hardware and software requirements are met before proceeding with the setup and execution of the project. The next section will guide you through the installation and setup process.

Installation and Setup:

In this section, we'll guide you through the process of setting up LangChain and the OpenAI API, both crucial components of this project. You will need to install some other packages also, we will discuss about those in the upcoming sections.

LangChain Setup:

To set up LangChain for this project, follow the steps below:

1. **Installation**: First, you need to install the LangChain package. It is typically installed via pip, a Python package manager. Open your terminal or command prompt and type the following command: `pip install langchain`
2. **Configuration**: After installation, you need to configure LangChain according to your specific requirements. This usually involves specifying your input and output formats, and any language-specific settings. Please refer to the LangChain documentation for more detailed instructions.

OpenAI API Setup:

Setting up the OpenAI API involves obtaining an API key and installing the OpenAI Python client. Follow these steps:

1. **Obtain API Key**: To use the OpenAI API, you first need to sign up on the OpenAI platform and obtain an API key. Once you have the key, make sure to keep it secure and do not share it with others.
2. **Install OpenAI Python Client**: Open your terminal or command prompt and type the following command to install the OpenAI Python client: `pip install openai`

PDF Data Extraction:

Understanding the PDF Extraction Process:

PDFs are a common file format that encapsulate a complete description of a fixed-layout flat document, including the text, fonts, graphics, and other information needed to display it. This project aims to extract data from a PDF document and convert this information into OpenAI embeddings using LangChain.

The process begins with opening the PDF document and reading its contents. This is achieved using the PyPDF2 library, which provides functionality to read and extract information from a PDF document. We then parse the document page by page and extract the textual content from each page.

The extracted text is subsequently cleaned and processed. Cleaning the text involves removing non-alphabetic characters, converting the text to lowercase, tokenizing the text into individual words, removing stopwords (commonly used words like 'the', 'a', 'an', etc. that do not carry much meaningful information), and lemmatizing the words (reducing them to their base or root form).

The text is then segmented into sections based on uppercase headings. Each section, along with its heading, is stored in a dictionary, which is then appended to a list. This list holds the data of the entire PDF document in a structured and processed form.

Code Overview:

Now let’s take a look at the coding to extract the text from a PDF file and convert it into embeddings.

Import packages:

```
import os
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import ElasticVectorSearch, Pinecone, Weaviate, FAISS
from langchain.chains.question_answering import load_qa_chain
from langchain.llms import OpenAI
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
import PyPDF2
# Download necessary NLTK data
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
```

The above section of the code involves importing various Python modules and packages required for the successful execution of the program. Here's a breakdown of these imports:

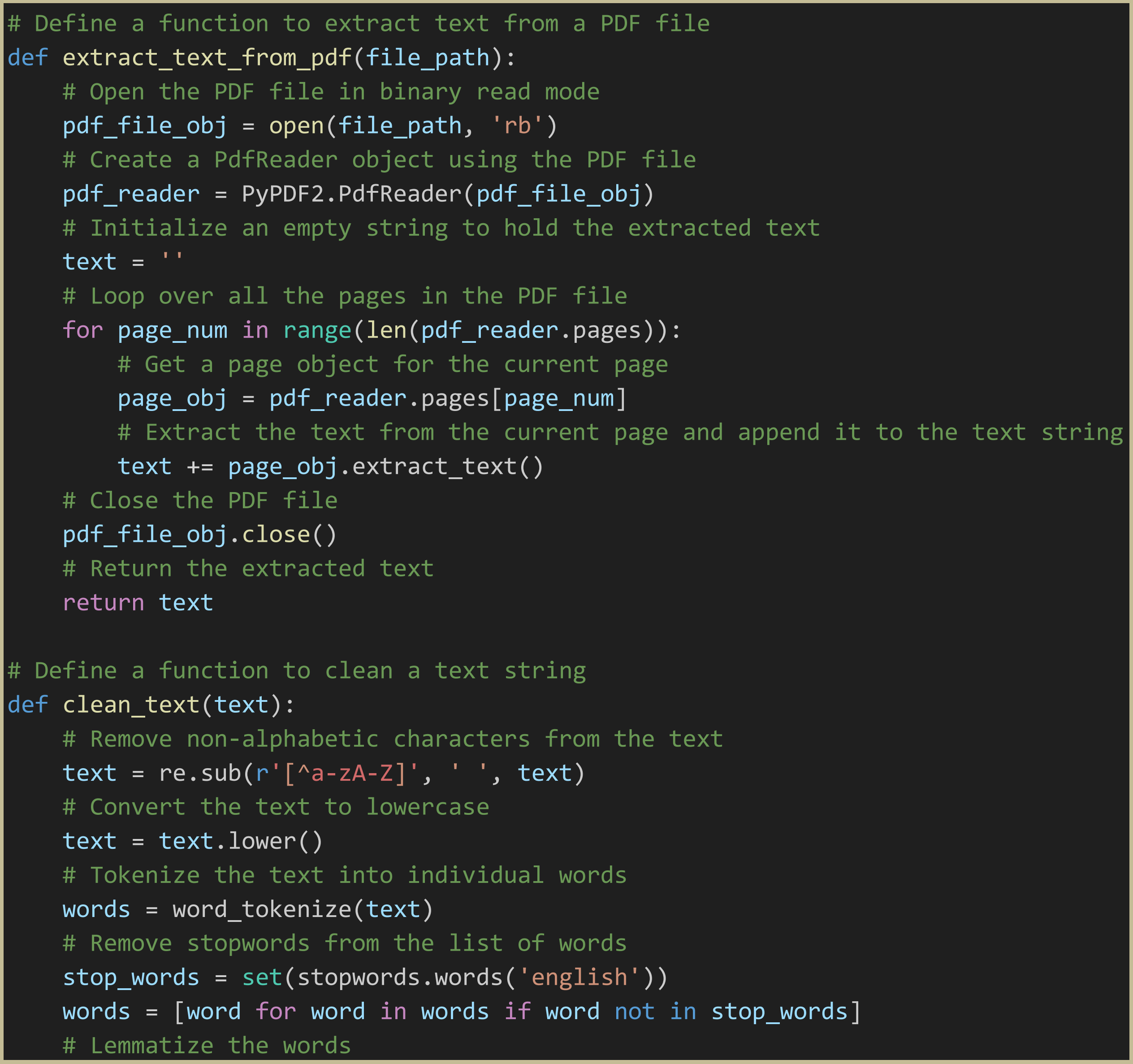
1. **os**: This built-in Python module is used for interacting with the operating system. It enables functionalities such as reading or writing to the file system, executing shell commands, and managing environment variables.
2. **OpenAIEmbeddings** from langchain.embeddings.openai: This module from the LangChain library facilitates the generation of OpenAI embeddings from the extracted text data.
3. **CharacterTextSplitter** from langchain.text_splitter: This module is responsible for splitting the extracted text into manageable pieces that can be processed by the LangChain and OpenAI systems.
4. **ElasticVectorSearch**, Pinecone, Weaviate, FAISS from langchain.vectorstores: These modules provide different methods for storing and retrieving the high-dimensional OpenAI embeddings. FAISS, for instance, is a library developed by Facebook AI for efficient similarity search and clustering of dense vectors.
5. **load_qa_chain** from langchain.chains.question_answering: This function loads a pre-trained question-answering system from LangChain, presumably for further processing of the OpenAI embeddings.
6. **OpenAI** from langchain.llms: This module from LangChain is used to interface with the OpenAI API.
7. **re**: This built-in Python module stands for Regular Expression, a powerful tool for manipulating text data. It is commonly used for tasks like searching, matching, and replacing patterns in text.
8. **nltk** and its related modules (stopwords, WordNetLemmatizer, word_tokenize): NLTK (Natural Language Toolkit) is a leading platform for building Python programs to work with human

language data. It provides easy-to-use interfaces for over 50 corpora and lexical resources. The specific modules imported are used for text preprocessing tasks such as tokenization, stop word removal, and lemmatization.

9. **PyPDF2**: This Python library is used for reading and extracting data from PDF files.

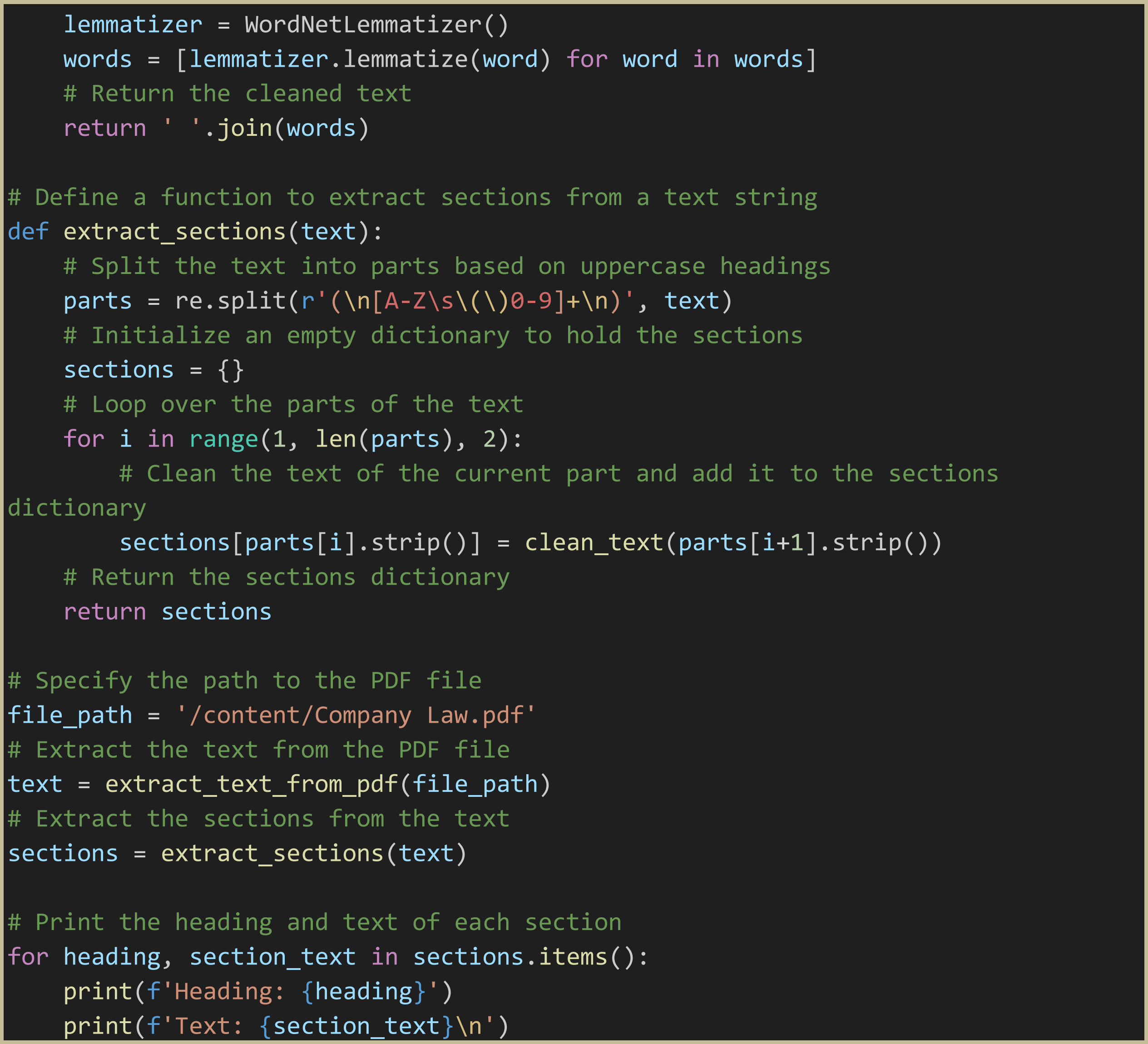
The last three lines are commands to download the necessary datasets from NLTK, including 'punkt' for tokenization, 'stopwords' for stop word lists, and 'wordnet' for lemmatization.

Extracting, Cleaning, and Segmenting Text from PDF Files:



```
# Define a function to extract text from a PDF file
def extract_text_from_pdf(file_path):
    # Open the PDF file in binary read mode
    pdf_file_obj = open(file_path, 'rb')
    # Create a PdfReader object using the PDF file
    pdf_reader = PyPDF2.PdfReader(pdf_file_obj)
    # Initialize an empty string to hold the extracted text
    text = ''
    # Loop over all the pages in the PDF file
    for page_num in range(len(pdf_reader.pages)):
        # Get a page object for the current page
        page_obj = pdf_reader.pages[page_num]
        # Extract the text from the current page and append it to the text string
        text += page_obj.extract_text()
    # Close the PDF file
    pdf_file_obj.close()
    # Return the extracted text
    return text

# Define a function to clean a text string
def clean_text(text):
    # Remove non-alphabetic characters from the text
    text = re.sub(r'[^a-zA-Z]', ' ', text)
    # Convert the text to lowercase
    text = text.lower()
    # Tokenize the text into individual words
    words = word_tokenize(text)
    # Remove stopwords from the list of words
    stop_words = set(stopwords.words('english'))
    words = [word for word in words if word not in stop_words]
    # Lemmatize the words
```



```
    lemmatizer = WordNetLemmatizer()
    words = [lemmatizer.lemmatize(word) for word in words]
    # Return the cleaned text
    return ' '.join(words)

# Define a function to extract sections from a text string
def extract_sections(text):
    # Split the text into parts based on uppercase headings
    parts = re.split(r'(\n[A-Z\s\(\)0-9]+\n)', text)
    # Initialize an empty dictionary to hold the sections
    sections = {}
    # Loop over the parts of the text
    for i in range(1, len(parts), 2):
        # Clean the text of the current part and add it to the sections
dictionary
        sections[parts[i].strip()] = clean_text(parts[i+1].strip())
    # Return the sections dictionary
    return sections

# Specify the path to the PDF file
file_path = '/content/Company Law.pdf'
# Extract the text from the PDF file
text = extract_text_from_pdf(file_path)
# Extract the sections from the text
sections = extract_sections(text)

# Print the heading and text of each section
for heading, section_text in sections.items():
    print(f'Heading: {heading}')
    print(f'Text: {section_text}\n')
```

Interpretation:

This section of the code defines three main functions: ***extract_text_from_pdf***, ***clean_text***, and ***extract_sections***, which are used for processing the text data extracted from PDF files.

extract_text_from_pdf(file_path):

This function is designed to extract all the text from a PDF file. It takes one argument, file_path, which is the path to the PDF file. Here's what each line does:

- ***pdf_file_obj = open(file_path, 'rb'):*** This line opens the PDF file in 'rb' (read binary) mode, because PDFs are binary files. The opened file is stored in the pdf_file_obj variable.

- ***pdf_reader*** = PyPDF2.PdfReader(pdf_file_obj): This line creates a PdfReader object from the PyPDF2 library, which is used to read the content of the PDF file.
- ***text*** = '': This line initializes an empty string that will be used to store the extracted text.
- ***for page_num in range(len(pdf_reader.pages))::*** This line starts a loop that iterates over the pages of the PDF file. The range(len(pdf_reader.pages)) part generates a sequence of numbers from 0 to one less than the number of pages in the PDF, effectively creating a list of all page numbers.
- ***page_obj = pdf_reader.pages[page_num]:*** This line gets the page object for the current page number.
- ***text += page_obj.extract_text():*** This line extracts the text from the current page and adds (concatenates) it to the text string.
- ***pdf_file_obj.close():*** This line closes the PDF file. It's good practice to close files after you're done with them to free up system resources.
- ***return text:*** Finally, this line returns the extracted text.

clean_text(text):

This function cleans a text string. It takes one argument, text, which is the string to be cleaned. Here's what each line does:

- ***text = re.sub(r'[^a-zA-Z]', ' ', text):*** This line uses a regular expression (regex) to replace all non-alphabetic characters in the text with spaces.
- ***text = text.lower():*** This line converts all the text to lowercase. This is done because in many contexts, the case of a word doesn't affect its meaning, and converting everything to lowercase can simplify subsequent processing.
- ***words = word_tokenize(text):*** This line uses the word_tokenize function from NLTK to split the text into individual words (tokens).
- ***stop_words = set(stopwords.words('english')):*** This line creates a set of English stopwords. Stopwords are common words like 'the', 'is', and 'and' that often don't carry much meaning and can be removed from the text.
- ***words = [word for word in words if word not in stop_words]:*** This line uses a list comprehension to create a new list that contains only the words that are not in the list of stopwords.
- ***lemmatizer = WordNetLemmatizer():*** This line creates a WordNetLemmatizer object. A lemmatizer is a tool that reduces words to their base or root form (their lemma). For example, it would convert 'running' to 'run'.

- ***words = [lemmatizer.lemmatize(word) for word in words]:*** This line uses another list comprehension to lemmatize all the words.
- ***return ' '.join(words):*** Finally, this line joins the words back together into a single string (with spaces in between the words) and returns it.

extract_sections(text):

This function splits a text string into sections based on uppercase headings. It takes one argument, text, which is the string to be split. Here's what each line does:

- ***parts = re.split(r'(\n[A-Z\s\(\)0-9]+\n)', text):*** This line uses a regular expression (regex) to split the text into parts. The regex pattern matches one or more uppercase letters, spaces, parentheses, or numbers that are on a line by themselves (preceded and followed by a newline character). The parentheses around the pattern mean that the matched headings are also included in the resulting list.
- ***sections = {}:*** This line initializes an empty dictionary to store the sections.
- ***for i in range(1, len(parts), 2)::*** This line starts a loop that iterates over the odd-numbered elements of the parts list. These are the headings, because the re.split method alternates between non-matching and matching parts.
- ***sections[parts[i].strip()] = clean_text(parts[i+1].strip()):*** This line adds a new entry to the sections dictionary. The key is the heading (with any leading or trailing whitespace removed), and the value is the text following the heading (cleaned using the clean_text function).
- ***return sections:*** Finally, this line returns the sections dictionary.

The last part of the code uses these functions to extract, clean, and split the text from a specific PDF file:

- ***file_path = '/content/Company Law.pdf':*** This line sets the file path of the PDF file to process.
- ***text = extract_text_from_pdf(file_path):*** This line calls the extract_text_from_pdf function to extract the text from the PDF file.
- ***sections = extract_sections(text):*** This line calls the extract_sections function to split the text into sections.
- ***for heading, section_text in sections.items()::*** This line starts a loop that iterates over the items in the sections dictionary. Each item is a pair of a heading and the corresponding section text.
- ***print(f'Heading: {heading}') and print(f'Text: {section_text}\n'):*** These lines print the heading and the cleaned section text.

Building Section Dictionary and Text Embedding:

This section of the code transforms the sectioned text into a format suitable for querying and information retrieval using the OpenAI API. It also sets up the environment for performing a semantic search in the text. Here is the code:

```
# Initialize an empty list to store section dictionaries
section_list = []

# Iterate over the sections dictionary
for heading, section_text in sections.items():
    # Create a dictionary for each section with "Heading" and "Text" as keys
    section_dict = {"Heading": heading, "Text": section_text}
    # Append the section dictionary to the section_list
    section_list.append(section_dict)

# Get the number of sections by finding the length of section_list
len(section_list)

# Print the dictionary for the second section in the list
section_list[1]

# Combine the heading and text of each section into a single string
# and store these strings in a new list, texts
texts = [section['Heading'] + ' ' + section['Text'] for section in section_list]

# Set the OpenAI API key as an environment variable
os.environ["OPENAI_API_KEY"] = "OpenAI API Key Here"

# Initialize a FAISS vector store with the texts and their corresponding
embeddings
vector_store = FAISS.from_texts(texts, embeddings)

# Load a question answering chain from OpenAI
chain = load_qa_chain(OpenAI(), chain_type='stuff')

# Define the query for the question you want to ask
query = 'Explain in detail: WINDING UP BY THE TRIBUNAL'

# Perform a similarity search in the vector store using the query
# This will return a list of documents (sections of text) that are most similar
to the query
docs = vector_store.similarity_search(query)
```

```
# Run the question answering chain on the documents with the query as the
question
# This will return the answer to the question based on the content of the
documents
chain.run(input_documents=docs, question=query)
```

Building Section Dictionary:

The first part of the code organizes the sections into a list of dictionaries for easier manipulation:

- ***section_list = []:*** This line initializes an empty list, section_list, that will store dictionaries. Each dictionary will represent a section of text, including its heading and body.
- ***for heading, section_text in sections.items()::*** This line starts a loop that iterates over each section (heading and text) in the sections dictionary.
- ***section_dict = {"Heading": heading, "Text": section_text}:*** This line creates a dictionary with two keys, "Heading" and "Text", and assigns the corresponding values from the current section.
- ***section_list.append(section_dict):*** This line appends the dictionary to the section_list.
- ***len(section_list):*** This line prints the length of the section_list, which is the number of sections.
- ***section_list[1]:*** This line prints the dictionary for the second section (Python list indices start at 0).

Preparing Texts for Embedding:

- ***texts = [section['Heading'] + ' ' + section['Text'] for section in section_list]:*** This line uses a list comprehension to create a new list, texts, that contains the full text (heading and body) of each section.

Setting up OpenAI API:

- ***os.environ["OPENAI_API_KEY"] = "OpenAI API Key Here":*** This line sets the environment variable OPENAI_API_KEY to your OpenAI API key. This key is used to authenticate your requests to the OpenAI API.

Initializing Vector Store:

- ***vector_store = FAISS.from_texts(texts, embeddings):*** This line initializes a FAISS vector store from the texts using the specified embeddings. A vector store is a structure that enables efficient similarity search in a high-dimensional space.

Loading Question Answering Chain:

- ***chain = load_qa_chain(OpenAI(), chain_type='stuff'):*** This line loads a question answering chain, which is a sequence of transformations that process input text and produce an answer to a question.

Asking Questions:

The final part of the code demonstrates how to ask a question and get an answer:

- ***query = 'Explain in detail: WINDING UP BY THE TRIBUNAL':*** This line sets the question to ask.
- ***docs = vector_store.similarity_search(query):*** This line performs a similarity search in the vector store using the query as the search key. The result is a list of documents (sections of text) that are most similar to the query.
- ***chain.run(input_documents=docs, question=query):*** This line runs the question answering chain on the documents, with the query as the question. The result is the answer to the question, based on the content of the documents.

Conclusion:

The task of converting PDF data into OpenAI embeddings using LangChain is an effective method to process and analyze unstructured data from PDF documents. Through this process, we are able to extract and clean text, segment it into meaningful sections, and transform it into a structured and analyzable format.

The PyPDF2 library provides a robust method for reading and extracting text from PDF documents, while the nltk library assists in cleaning and processing the text. The utilization of regular expressions allows for efficient segmentation of the text into sections based on uppercase headings.

Once the data has been extracted and processed, we utilize the OpenAI API for further analysis. The LangChain's OpenAIEmbeddings and FAISS classes allow us to create embeddings of the text and initialize a vector store, which is used to perform similarity searches and answer queries about the document's content.

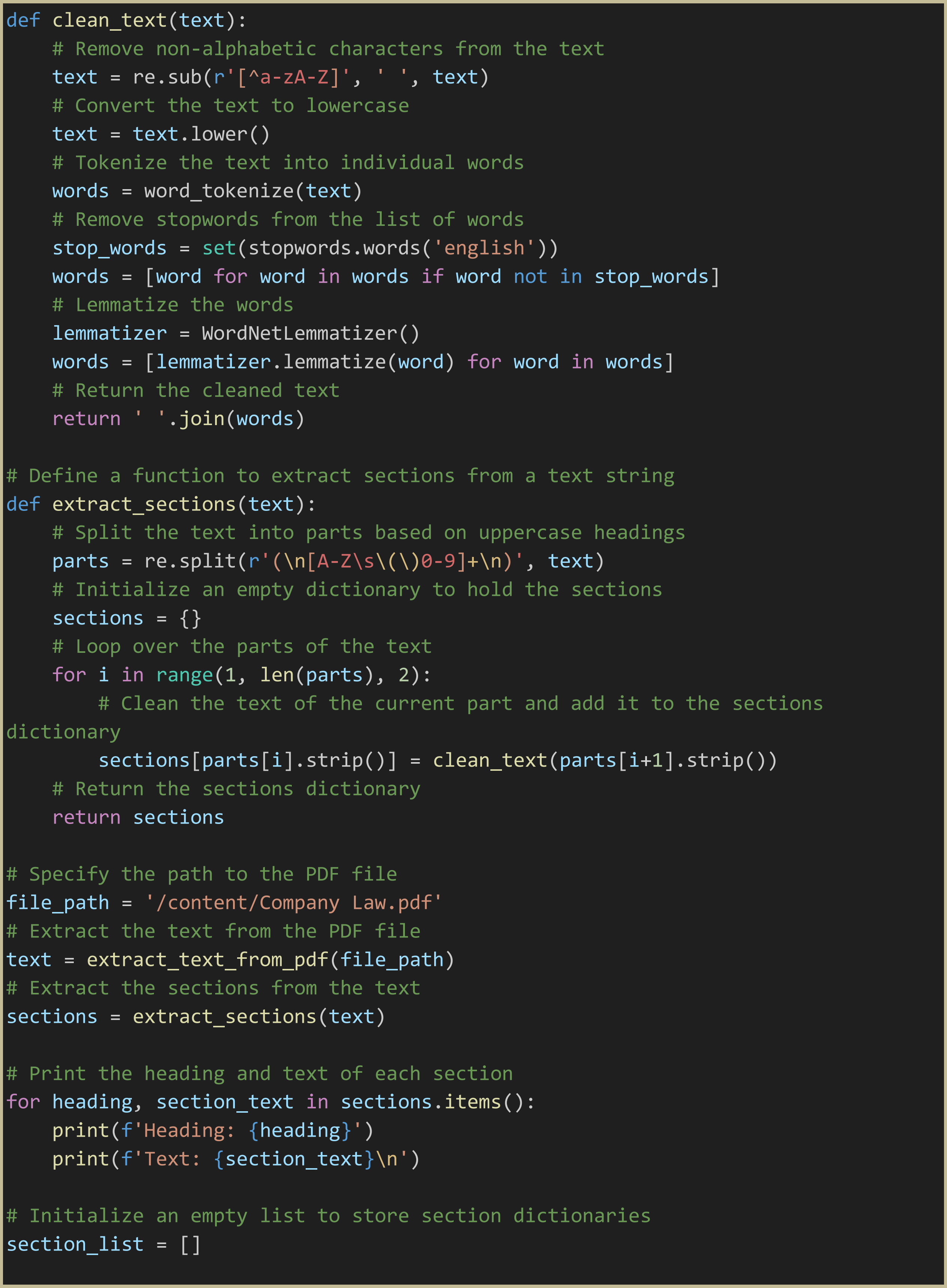
This process demonstrates the power and versatility of natural language processing and machine learning techniques in extracting insights from unstructured data. Through this project, we have transformed a static, unstructured PDF document into a dynamic, structured, and queryable resource, demonstrating the potential for advanced data extraction and analysis methods.

By applying this process to other documents, we can build a rich and searchable knowledge base, making it easier to find and extract information. This has numerous applications in fields like data mining, information retrieval, and AI-powered document analysis, among others. The combination of OpenAI, LangChain, and Python libraries for data processing and analysis provides a powerful toolset for these tasks.

Code:

```
# Define a function to extract text from a PDF file
def extract_text_from_pdf(file_path):
    # Open the PDF file in binary read mode
    pdf_file_obj = open(file_path, 'rb')
    # Create a PdfReader object using the PDF file
    pdf_reader = PyPDF2.PdfReader(pdf_file_obj)
    # Initialize an empty string to hold the extracted text
    text = ''
    # Loop over all the pages in the PDF file
    for page_num in range(len(pdf_reader.pages)):
        # Get a page object for the current page
        page_obj = pdf_reader.pages[page_num]
        # Extract the text from the current page and append it to the text string
        text += page_obj.extract_text()
    # Close the PDF file
    pdf_file_obj.close()
    # Return the extracted text
    return text

# Define a function to clean a text string
```



```python
def clean_text(text):
    # Remove non-alphabetic characters from the text
    text = re.sub(r'[^a-zA-Z]', ' ', text)
    # Convert the text to lowercase
    text = text.lower()
    # Tokenize the text into individual words
    words = word_tokenize(text)
    # Remove stopwords from the list of words
    stop_words = set(stopwords.words('english'))
    words = [word for word in words if word not in stop_words]
    # Lemmatize the words
    lemmatizer = WordNetLemmatizer()
    words = [lemmatizer.lemmatize(word) for word in words]
    # Return the cleaned text
    return ' '.join(words)

# Define a function to extract sections from a text string
def extract_sections(text):
    # Split the text into parts based on uppercase headings
    parts = re.split(r'(\n[A-Z\s\(\)0-9]+\n)', text)
    # Initialize an empty dictionary to hold the sections
    sections = {}
    # Loop over the parts of the text
    for i in range(1, len(parts), 2):
        # Clean the text of the current part and add it to the sections
dictionary
        sections[parts[i].strip()] = clean_text(parts[i+1].strip())
    # Return the sections dictionary
    return sections

# Specify the path to the PDF file
file_path = '/content/Company Law.pdf'
# Extract the text from the PDF file
text = extract_text_from_pdf(file_path)
# Extract the sections from the text
sections = extract_sections(text)

# Print the heading and text of each section
for heading, section_text in sections.items():
    print(f'Heading: {heading}')
    print(f'Text: {section_text}\n')

# Initialize an empty list to store section dictionaries
section_list = []
```

```python
# Iterate over the sections dictionary
for heading, section_text in sections.items():
    # Create a dictionary for each section with "Heading" and "Text" as keys
    section_dict = {"Heading": heading, "Text": section_text}
    # Append the section dictionary to the section_list
    section_list.append(section_dict)

# Get the number of sections by finding the length of section_list
len(section_list)

# Print the dictionary for the second section in the list
section_list[1]

# Combine the heading and text of each section into a single string
# and store these strings in a new list, texts
texts = [section['Heading'] + ' ' + section['Text'] for section in section_list]

# Set the OpenAI API key as an environment variable
os.environ["OPENAI_API_KEY"] = "OpenAI API Key"

# Initialize a FAISS vector store with the texts and their corresponding embeddings
vector_store = FAISS.from_texts(texts, embeddings)

# Load a question answering chain from OpenAI
chain = load_qa_chain(OpenAI(), chain_type='stuff')

# Define the query for the question you want to ask
query = 'Explain in detail: WINDING UP BY THE TRIBUNAL'

# Perform a similarity search in the vector store using the query
# This will return a list of documents (sections of text) that are most similar to the query
docs = vector_store.similarity_search(query)

# Run the question answering chain on the documents with the query as the question
# This will return the answer to the question based on the content of the documents
chain.run(input_documents=docs, question=query)
```