Assignment 1

Mehrshad Kafi

Contents

ChatPDF

Testing the Application **2**
Report **2**
Implement PDF Parsing Mechanism **4**
Scanned Document 5
LangChain Integration for Conversational Interface **6**
Web Interface **10**
Python Flask Application - Backend (app.py) 10
HTML Frontend (templates/index.html) 12

Snoop Dogg Persona

Testing the Application **16**
Report **16**
Fine-Tune the LLM **16**
Setting Up the Environment 16
Loading and Preparing the Dataset 16
Tokenization 17
Model Training 17
Model Evaluation and Text Generation 18
TTS System **19**
Preparation and Setup 20
Defining Text for Speech Conversion 20
Making a Request to the Text-to-Speech API 21
Saving the Generated Speech to a File 21
Playing the Audio File 21
Web Interface **21**
Python Flask Application - Backend (app.py) 22
HTML Frontend (templates/index.html) 24

ChatPDF

Testing the Application

To test the application:

- Extract the ChatPDF zip file and in the extracted folder run the terminal.
- Run the Flask application by executing `python app.py` in the terminal.
- Open the web browser and navigate to http://127.0.0.1:5000/.
- Try uploading a PDF and querying information from it.

Report

Developing an application that allows users to upload PDF documents and engage in natural language conversations to query information from the document involves several key steps and components. Let's break down the objective and requirements to create a clear development roadmap:

Step 1: Install Required Libraries:
- PDF Parsing: PyMuPDF or PDFMiner
- Natural Language Processing: Install LangChain or a similar open-source language model library.

```
pip install flask
pip install pymupdf  # Replace pymupdf with pdfminer.six if preferred
pip install langchain
pip install openai
pip install faiss-cpu
pip install tiktoken
pip install -U langchain-openai
```

Step 2: Implement PDF Parsing Mechanism

PyMuPDF:
- Ideal for extracting text as it's fast and easy to use.
- Use `fitz` (PyMuPDF's interface) to open and read PDF files, extracting text.

PDFMiner:

- More flexible in text extraction and analysis.
- Provides detailed location and font information, suitable for complex layouts.

Choose based on our needs. If speed and simplicity are key, we go with PyMuPDF. If we need detailed text analysis, PDFMiner is a better choice.

Step 3: Integrate Language Model for Conversational Interface

LangChain:
- An excellent tool for integrating language models into applications.
- Explore LangChain documentation to understand how to utilize it for creating conversational interfaces that can interpret user queries and generate responses based on the PDF content.

Step 4: Develop Web Interface for User Interaction

Frontend:
- Create a simple web interface using HTML, CSS, and JavaScript.
- Include an upload button for PDFs and a text input for queries.

Backend:
- Use Flask or Django (for Python) to handle file uploads and serve the frontend.
- Implement endpoint(s) that accept PDF uploads and queries, process them, and return responses.

Step 5: Handling User Queries and Extracting Answers

- Upon receiving a user query, use the parsed PDF content and LangChain to understand the query context and find relevant information within the document.
- Return the extracted information as a response to the user.

Step 6: Testing and Refinement

- Test the application with various PDFs to ensure accurate text extraction and response generation.
- Refine the conversational model as needed to improve understanding and response accuracy.

Now, let's delve into more details for each step:

Implement PDF Parsing Mechanism

PyMuPDF is a Python binding for MuPDF, a lightweight PDF and XPS viewer. It's fast and easy to use for extracting text from PDFs. Here's a simple guide to get we started:

Step 1: Extract Text from Each Page

Iterate through the pages of the PDF document and extract the text. PyMuPDF allows to extract the entire page's text with a single method call.

```
def extract_text_from_pdf(pdf_path):
    doc = fitz.open(pdf_path)
    text = ""

    for page in doc:
        # Extracting text from each page
        text += page.get_text()

    doc.close()  # Remember to close the document
    return text
```

Step 2: Use the Extracted Text

Now, we have the extracted text from the PDF document. We can print it or return it from a function to use in your application.

```
pdf_path = 'path/to/your/document.pdf'
# Make sure to replace this with your PDF's path
extracted_text = extract_text_from_pdf(pdf_path)
print(extracted_text)
```

This simple example shows how to extract text from a PDF file using PyMuPDF. We can extend this by implementing more complex logic to handle the extracted text, such as parsing it into structured data or searching for specific information based on user queries.

Remember, the quality of the text extraction can vary based on the PDF's contents and how it was created. Some PDFs might contain text in images or use fonts and layouts that are difficult to extract accurately. In those cases, additional processing, like OCR (Optical Character Recognition), may be necessary.

Scanned Document

If the PDF is a scan of a document, we're essentially dealing with images of text, making OCR (Optical Character Recognition) the appropriate technology to extract text from these pages. The process involves converting each page of the PDF into an image and then applying OCR to these images to extract the text. Below is a detailed guide on how to do this using PyMuPDF for handling PDFs, PIL (Python Imaging Library) for image processing, and pytesseract, a Python wrapper for Tesseract OCR, which is one of the most accurate and widely used OCR engines.

Step 1: Install the Necessary Libraries

We'll need to install PyMuPDF, PIL (or Pillow, the friendly PIL fork), and pytesseract. If you haven't installed Tesseract OCR itself, you'll need to do that as well.

```
pip install pymupdf Pillow pytesseract
```

- Install Tesseract OCR: The installation process varies depending on our operating system. On Ubuntu, we can install it using:

```
  sudo apt-get install tesseract-ocr
```

For other operating systems, follow the instructions on the [Tesseract GitHub page](https://github.com/tesseract-ocr/tesseract) or relevant package managers.

Step 2: Extracting Images and Applying OCR

This Python script opens a scanned PDF, extracts each page as an image, and applies OCR to extract text from these images.

```
import fitz  # Import PyMuPDF
from PIL import Image
import pytesseract
import io

def extract_text_from_scanned_pdf(pdf_path):
    doc = fitz.open(pdf_path)
    extracted_text = ""

    for page_num in range(len(doc)):
```

```
        # Get the page
        page = doc.load_page(page_num)

        # Extract the image of the page
        pix = page.get_pixmap()
        img_bytes = pix.tobytes("ppm")

        # Convert bytes to an image
        image = Image.open(io.BytesIO(img_bytes))

        # Apply OCR to extract text
            page_text = pytesseract.image_to_string(image, lang='eng')  # Specify the
language as needed
        extracted_text += page_text + "\n"

    doc.close()
    return extracted_text

pdf_path = 'path/to/your/scanned/document.pdf'
extracted_text = extract_text_from_scanned_pdf(pdf_path)
print(extracted_text)
```

Note:
- Language Option: The `lang='eng'` argument specifies that the text to be extracted is in English. If your document is in another language, you can change this parameter to the appropriate ISO 639-2/T language code (e.g., 'spa' for Spanish, 'fra' for French, etc.). You may need to download additional language data for Tesseract for languages other than English.
- Image Quality and OCR Accuracy: The quality of OCR results can vary significantly based on the quality of the scanned images. Poorly scanned documents may require image preprocessing techniques (e.g., noise reduction, contrast adjustment) to improve OCR accuracy.

This approach should allow us to extract text from scanned PDF documents, turning images of text back into searchable, editable text. Remember, OCR isn't perfect and might require some manual corrections for best results, especially with complex layouts or lower-quality scans.

LangChain Integration for Conversational Interface

For integrating a language model using LangChain to create a conversational interface that can understand and generate responses based on PDF content, the following steps and example code can be followed.

```
!pip install langchain
!pip install openai
!pip install faiss-cpu
!pip install tiktoken
!pip install -U langchain-openai
```

Here's a breakdown of what each command does:

1. !pip install langchain:
This command installs the `langchain` package. LangChain is a Python library designed to facilitate the integration of large language models (LLMs) into applications, providing tools for building conversational interfaces, summarization, question-answering systems, and more.

2. !pip install openai:
This command installs the `openai` package, which is the official Python client library for the OpenAI API. This library allows you to interact with OpenAI's language models, such as GPT-3, Codex, and others, for tasks including text generation, code generation, and more. It requires an API key from OpenAI to use.

3. !pip install faiss-cpu:
This command installs the `faiss-cpu` package. FAISS (Facebook AI Similarity Search) is a library developed by Facebook AI Research for efficient similarity search and clustering of dense vectors. The `faiss-cpu` version is specifically optimized for use on CPU hardware. It's useful in scenarios where you need to find items similar to a query item in a large dataset, such as searching through embeddings generated by language models.

4. !pip install tiktoken:
This command installs the `tiktoken` package. tiktoken is a fast open-source tokenizer by OpenAI. Given a text string (e.g., "tiktoken is great!") and an encoding (e.g., "cl100k_base"), a tokenizer can split the text string into a list of tokens (e.g., ["t", "ik", "token", " is", " great", "!"]).

5. !pip install -U langchain-openai:
This command installs or updates (`-U` flag for update) the `langchain-openai` package. This package suggests an extension or integration of LangChain specifically designed to work with OpenAI's APIs, likely providing tools or wrappers that facilitate the use of OpenAI's language models within the LangChain framework. This could simplify tasks such as embedding LangChain's functionality with OpenAI's models for building conversational interfaces or other language-based applications.

```
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS
```

```
from langchain_openai import OpenAIEmbeddings
```

CharacterTextSplitter from langchain.text_splitter: A utility for dividing text into segments based on character count. This can be particularly useful for processing large texts or documents in scenarios where input size to a language model is limited. By splitting texts into manageable sizes, it ensures that the processing of large documents is feasible without losing content due to input size restrictions.

FAISS from langchain.vectorstores: An efficient similarity search and clustering of dense vectors library, which is optimized for CPUs in this context. FAISS is designed to enable fast vector similarity lookup, which is essential for tasks such as finding similar documents, performing clustering of text embeddings, or any other task where the similarity between vectors needs to be quantified quickly and efficiently. The integration with LangChain suggests its use for managing vector representations of text, likely generated by language models for various applications like semantic search, document clustering, or nearest neighbor queries in natural language processing tasks.

OpenAIEmbeddings from langchain_openai: This module facilitates the generation of text embeddings using OpenAI's models. Embeddings are dense vector representations of text that capture semantic meaning. Generating embeddings with OpenAI's models could leverage the advanced capabilities of models like GPT-3 to understand and represent the nuances of text. This is crucial for applications that require understanding the content at a deeper level, such as semantic search, content recommendation, and more sophisticated natural language understanding tasks.

```
text_splitter = CharacterTextSplitter(
    separator = "\n",
    chunk_size = 800,
    chunk_overlap  = 200,
    length_function = len,
)
texts = text_splitter.split_text(extracted_text)

import os
os.environ["OPENAI_API_KEY"] = "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"
embeddings = OpenAIEmbeddings()

document_search = FAISS.from_texts(texts, embeddings)

from langchain.chains.question_answering import load_qa_chain
from langchain.llms import OpenAI
```

```
chain = load_qa_chain(OpenAI(), chain_type="stuff")

query = "What is the topic of this document?"
docs = document_search.similarity_search(query)
chain.run(input_documents=docs, question=query)
```

Text Splitting

CharacterTextSplitter: This is initialized with specific parameters to split large texts into manageable chunks. The `separator` defines the boundary between text chunks (here, a newline character), `chunk_size` sets the target length for each chunk (800 characters), `chunk_overlap` allows chunks to overlap by 200 characters to preserve context across boundaries, and `length_function` (here, the built-in `len` function) measures the length of text chunks. This splitter then processes `extracted_text` to produce smaller, more manageable pieces of text.

Embedding Generation and Environment Setup

Environment Variable for OpenAI API Key: The script sets an environment variable for the OpenAI API key, which is necessary for authenticating requests to OpenAI's services, including generating embeddings or utilizing their language models. It's crucial to keep such keys private and secure.

OpenAIEmbeddings: An instance is created to generate embeddings for texts, which are dense vector representations capturing the semantic essence of the texts.

Document Search with FAISS

FAISS.from_texts: This line creates a FAISS vector store from the text chunks and their corresponding embeddings. FAISS is used here for its efficient similarity search capabilities, allowing for rapid retrieval of documents similar to a given query based on their vector representations.

Question Answering Chain

Loading a QA Chain: The code loads a question-answering (QA) chain with LangChain, using OpenAI's language models. The `chain_type` is specified as "stuff". The stuff documents chain ("stuff" as in "to stuff" or "to fill") is the most straightforward of the document chains. It takes a list of documents, inserts them all into a prompt and passes that prompt to an LLM. This chain is designed to handle QA tasks by leveraging the capabilities of large language models.

Query Processing

Executing a Query: A query ("What is the topic of this document?") is processed by first finding similar documents using the FAISS document search. The `similarity_search` function takes the query, likely converts it into an embedding, and retrieves documents (text chunks) that are semantically similar to the query.

Running the QA Chain: The QA chain is then run with the input documents and the query. This step likely involves generating answers based on the content of the similar documents and the specific question asked.

Web Interface

Here we are developing a simple web interface using Flask for the backend. This interface will enable users to upload PDF files and submit queries, which will be processed by already implemented PDF parsing and conversation model. Let's put together the complete Python Flask application code and the corresponding HTML code for the frontend. This setup will allow users to upload a PDF file, then immediately enter queries to get information from the uploaded document.

Python Flask Application - Backend (app.py)

```
# Import necessary modules from Flask and other libraries
from flask import Flask, request, render_template, jsonify, session
import os
import fitz  # PyMuPDF for PDF processing

# Initialize the Flask application
app = Flask(__name__)

# Configuration for file uploads
UPLOAD_FOLDER = 'uploads'  # Directory where uploaded files will be stored
ALLOWED_EXTENSIONS = {'pdf'}  # Allowed file types for upload
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER  # Apply the upload folder configuration to
the app

# Function to check if the uploaded file has an allowed extension
def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

# Function to extract text from a PDF file using PyMuPDF
```

```
def extract_text_from_pdf(pdf_path):
    doc = fitz.open(pdf_path)  # Open the PDF file
    text = ""
    for page in doc:  # Iterate through each page in the PDF
        text += page.get_text()  # Extract text from the page and concatenate it
    doc.close()  # Close the PDF file after processing
    return text

# Placeholder function for processing queries against extracted text
def answer_query(extracted_text, query):
    # Placeholder implementation. Should be replaced with actual logic. Explained
before
    Return response

# Route for the index page
@app.route('/')
def index():
    return render_template('index.html')  # Serve the index.html template

# Route to handle file uploads
@app.route('/upload', methods=['POST'])
def upload_file():
    # Check if the request has the file part
    if 'file' not in request.files:
        return jsonify({'error': 'No file part'}), 400
    file = request.files['file']  # Get the file from the request
    # Check if the file is valid
    if file.filename == '' or not allowed_file(file.filename):
        return jsonify({'error': 'No selected file or file type not allowed'}), 400

    filename = os.path.join(app.config['UPLOAD_FOLDER'], file.filename)  # Determine
the file save path
    file.save(filename)  # Save the file
    extracted_text = extract_text_from_pdf(filename)  # Extract text from the saved PDF
    session['extracted_text'] = extracted_text  # Store the extracted text in session
    return jsonify({'message': 'File successfully uploaded'}), 200  # Return success
message

# Route to handle queries
@app.route('/query', methods=['POST'])
def handle_query():
    query = request.json.get('query', '')  # Get the query from the request
    # Check if there is a query and if text has been extracted from a PDF
```

```
    if not query or 'extracted_text' not in session:
        return jsonify({'error': 'Empty query or no document uploaded'}), 400
    extracted_text = session['extracted_text']  # Retrieve the extracted text from
session
    response = answer_query(extracted_text, query)  # Generate a response based on the
query
    return jsonify({'answer': response}), 200  # Return the response

# Main entry point of the application
if __name__ == '__main__':
    # Ensure the upload directory exists
    if not os.path.exists(UPLOAD_FOLDER):
        os.makedirs(UPLOAD_FOLDER)
    app.run(debug=True)  # Start the Flask application with debugging enabled
```

HTML Frontend (templates/index.html)

This HTML file should be placed in a folder named templates within the project directory.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>ChatPDF Interface</title>
    <style>
        /* Basic CSS for styling the page, inputs, buttons, and containers */
        body {
            font-family: Arial, sans-serif; /* Sets the font for the body */
            margin: 0;
            padding: 0;
            background-color: #f0f0f0; /* Light gray background */
            text-align: center; /* Centers text */
        }
        .container {
            width: 50%; /* Container width */
            margin: auto; /* Centers the container horizontally */
            background-color: #fff; /* White background for the container */
            padding: 20px; /* Padding inside the container */
            margin-top: 30px; /* Margin at the top of the container */
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1); /* Subtle shadow around the
container */
```

```
        }
        h1 {
            color: #333; /* Dark grey color for the heading */
        }
        input[type="file"] {
            margin: 20px 0; /* Margin around the file input */
        }
        input[type="submit"], button {
            background-color: #4CAF50; /* Green background for buttons */
            color: white; /* White text on buttons */
            padding: 10px 20px; /* Padding inside buttons */
            margin: 20px 0; /* Margin around buttons */
            border: none; /* No border for a cleaner look */
            cursor: pointer; /* Pointer cursor on hover */
        }
        input[type="submit"]:hover, button:hover {
            background-color: #45a049; /* Darker green on hover */
        }
        #querySection {
            margin-top: 20px; /* Margin above the query section */
        }
        input[type="text"] {
            width: 60%; /* Width of the text input */
            padding: 10px; /* Padding inside the text input */
            margin: 10px 0; /* Margin around the text input */
            display: inline-block; /* Allows the input to sit next to other inline
elements */
            border: 1px solid #ccc; /* Light grey border */
            border-radius: 4px; /* Rounded corners */
            box-sizing: border-box; /* Border and padding included in width */
        }
    </style>
</head>
<body>
    <div class="container">
        <!-- Main container for content -->
        <h1>ChatPDF created by Mehrshad Kafi</h1> <!-- Title and creator -->
        <form id="uploadForm" enctype="multipart/form-data">
            <!-- Form for uploading PDF files -->
            <input type="file" name="file" accept=".pdf"> <!-- File input for PDFs only
-->
            <input type="submit" value="Upload PDF"> <!-- Submit button for the form
-->
```

```
        </form>
        <p id="uploadResponse"></p> <!-- Placeholder for upload response message -->

        <div id="querySection" style="display:none;">
            <!-- Hidden section for queries, shown after upload -->
            <input type="text" id="queryInput" placeholder="Enter your query here">
<!-- Input for typing queries -->
            <button onclick="submitQuery()">Submit Query</button> <!-- Button to submit
the query -->
        </div>
        <p id="queryResponse"></p> <!-- Placeholder for query response message -->
    </div>

    <script>
        // JavaScript to handle the form submission without reloading the page
        document.getElementById('uploadForm').onsubmit = function(event) {
            event.preventDefault(); // Prevents the default form submission
            var formData = new FormData(this); // Prepares form data for submission
            fetch('/upload', {
                method: 'POST',
                body: formData,
            })
            .then(response => response.json()) // Parses the JSON response
            .then(data => {
                document.getElementById('uploadResponse').textContent = data.message;
// Displays upload response
                document.getElementById('querySection').style.display = 'block'; //
Shows the query section
            })
            .catch(error => {
                console.error('Error:', error); // Logs any errors
                document.getElementById('uploadResponse').textContent = 'Failed to
upload file.';
            });
        };

        function submitQuery() {
            // Function to handle query submissions
            const query = document.getElementById('queryInput').value; // Gets the
query from the input
            fetch('/query', {
                method: 'POST',
                headers: {'Content-Type': 'application/json'},
```

```
                body: JSON.stringify({query}), // Sends the query as JSON
            })
            .then(response => response.json()) // Parses the JSON response
            .then(data => {
                document.getElementById('queryResponse').textContent = data.answer; //
Displays the query response
            })
            .catch(error => {
                console.error('Error:', error); // Logs any errors
            });
        }
    </script>
</body>
</html>
```

Snoop Dogg Persona

Testing the Application

To test the application:

- Extract the ChatPDF zip file and in the extracted folder run the terminal.
- Run the Flask application by executing `python app.py` in the terminal.
- Open the web browser and navigate to http://127.0.0.1:5000/.
- Try uploading a PDF and querying information from it.

Report

The provided code demonstrates the process of fine-tuning a GPT-2 model on a dataset containing lyrics by Snoop Dogg, using Hugging Face's `transformers` and `datasets` libraries. This process involves several key steps, each contributing to the development of a language model tailored to generate text in the style of Snoop Dogg.

Fine-Tune the LLM

Setting Up the Environment

- The `transformers`, `datasets`, and `torch` libraries are installed, equipping the environment with the necessary tools for model training and manipulation of datasets.

```
!pip install transformers datasets torch
```

Loading and Preparing the Dataset

- A dataset featuring Snoop Dogg's lyrics is fetched using `load_dataset` from the `datasets` library, demonstrating the accessibility of curated datasets for model training.
- The dataset is split into training and validation sets, ensuring that there's a portion of the data reserved for evaluating the model's performance. This split is conducted with a 90:10 ratio, standard practice for training machine learning models.
- The GPT-2 tokenizer is loaded and configured to use its end-of-sequence token as a padding token, preparing the tokenizer for processing the text data. This adjustment allows for the

handling of sequences of varying lengths during training.

```
from datasets import load_dataset

# Assuming you've already loaded your dataset
dataset = load_dataset("huggingartists/snoop-dogg")

# Split the dataset into training and validation sets
train_test_split = dataset["train"].train_test_split(test_size=0.1)
```

Tokenization

- A tokenization function is defined and then applied to both the training and validation datasets. This process converts the raw text into a format that the model can understand, specifically, a sequence of token IDs. The function ensures that each text entry is padded to a fixed length and truncated if necessary, maintaining uniformity in sequence length.
- The tokenized datasets are further processed to associate the input IDs with labels, a necessary step for training the model on a language modeling task. This setup implies that the model will learn to predict the next token in a sequence, a common approach for training generative language models.

```
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
tokenizer.pad_token = tokenizer.eos_token  # Set padding token

def tokenize_function(examples):
    # Tokenize the texts and map the tokens to their IDs
    return tokenizer(examples["text"], padding="max_length", truncation=True,
max_length=128, return_tensors="pt")

# Apply tokenization to the dataset
tokenized_datasets = train_test_split.map(tokenize_function, batched=True)

# Make sure the labels are correctly set for language modeling
tokenized_datasets = tokenized_datasets.map(lambda examples: {'labels':
examples['input_ids']
```

Model Training

- A GPT-2 model pre-loaded with weights is fetched and prepared for fine-tuning. This step leverages the power of transfer learning, where a model pre-trained on a large corpus is adapted to a specific task or style, in this case, generating text in the style of Snoop Dogg.
- Training arguments are defined, including the number of epochs, batch sizes for training and evaluation, and the strategy for evaluation. These parameters guide the training process, influencing the model's learning rate and performance evaluation.
- A `Trainer` object is instantiated with the model, training arguments, and datasets. This object orchestrates the training process, managing data loading, model updating, and evaluation.
- The training process is executed, culminating in a model fine-tuned on the Snoop Dogg lyrics dataset. The output includes logs of training and validation losses, offering insights into the model's learning progress over epochs.

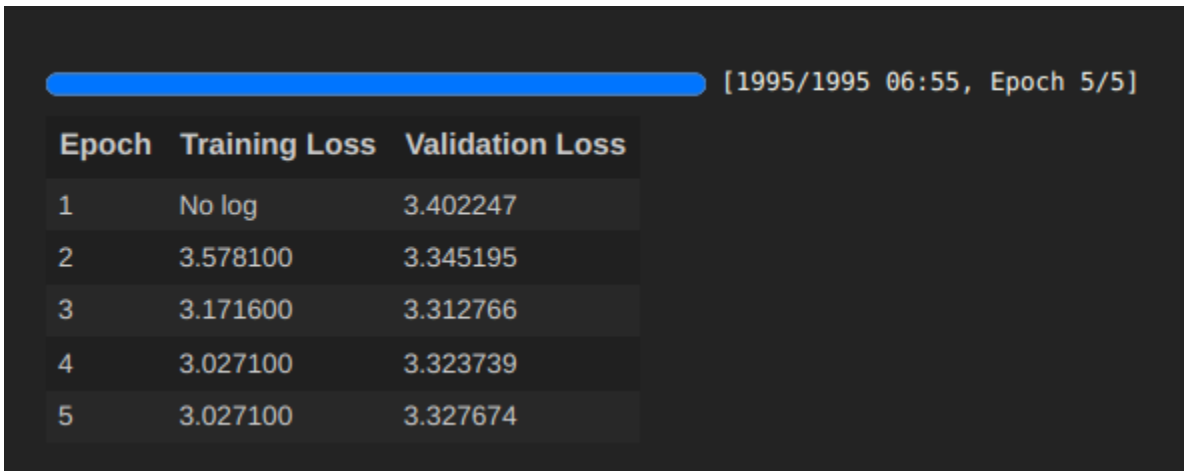
```
from transformers import GPT2LMHeadModel, Trainer, TrainingArguments

model = GPT2LMHeadModel.from_pretrained('gpt2')

training_args = TrainingArguments(
    output_dir='./results',          # Output directory
    num_train_epochs=5,              # Number of training epochs
    per_device_train_batch_size=4,   # Batch size per device during training
    per_device_eval_batch_size=4,    # Batch size for evaluation
    evaluation_strategy="epoch",     # Evaluate at the end of each epoch
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets['train'],
    eval_dataset=tokenized_datasets['test'],  # Note: 'test' here is the validation set
we created
)

trainer.train()
```

Model Evaluation and Text Generation

- After training, the model is saved for future use, encapsulating the fine-tuning results in a reusable format.
- A text generation pipeline is created with the fine-tuned model, enabling the generation of text based on prompts.
- A specific prompt is fed into the generator, resulting in a piece of generated text. This output demonstrates the model's ability to produce text in the style of Snoop Dogg, reflecting the

learning achieved through fine-tuning.

This code encapsulates the end-to-end process of adapting a pre-trained language model to a specific stylistic task, showcasing the capabilities of modern NLP frameworks and methodologies.



Epoch	Training Loss	Validation Loss
1	No log	3.402247
2	3.578100	3.345195
3	3.171600	3.312766
4	3.027100	3.323739
5	3.027100	3.327674

```
trainer.save_model("snoop-dogg-model")
```

```
from transformers import pipeline

generator = pipeline('text-generation', model="snoop-dogg-model", tokenizer='gpt2')

# prompt = "Just chillin', you know how we do, "
prompt = "how do you wake up my child"
texts = generator(prompt, max_length=100, num_return_sequences=1)

generated_text = ""
for text in texts:
    print(text['generated_text'])
    generated_text += text['generated_text']
```

TTS System

The provided Python script demonstrates how text is converted into speech using an API and then played back as audio. The process involves several key steps, each facilitated by the use of specific Python libraries and external resources.

```
import requests
import json
```

```
import os

text_to_speak = "This is the text you want to convert to speech."

response = requests.request(
  method="POST",
  url="https://api.neets.ai/v1/tts",
  headers={
    "Content-Type": "application/json",
    "X-API-Key": "c0d9aeeef6f14e98ad46c8d23e24535b"
  },
  json={
    "text": generated_text,
    "voice_id": "snoop-dogg",
    "params": {
      "model": "ar-diff-50k"
    }
  }
)

with open("snoop_dogg.mp3", "wb") as f:
  f.write(response.content)

audio_file = "snoop_dogg.mp3"
os.system("mpg321 " + audio_file)
```

Preparation and Setup

- The `requests`, `json`, and `os` libraries are imported, equipping the script with capabilities for making HTTP requests, handling JSON data, and interacting with the operating system, respectively.

Defining Text for Speech Conversion

- A string of text intended to be converted into speech is stored in the variable `text_to_speak`. This string serves as the input for the text-to-speech (TTS) process.

Making a Request to the Text-to-Speech API

- An HTTP POST request is made to the `https://api.neets.ai/v1/tts` endpoint, indicating the intent to use the Neets.ai TTS service. This request is crafted using the `requests` library.
- Headers specifying the content type as `application/json` and including an API key for authentication are included in the request. The API key is a placeholder and should be replaced with a valid key for actual use.
- The body of the request, formatted as JSON, contains the text to be converted, the desired voice identifier (`voice_id`), and model parameters. These details instruct the API on how to process the text and generate speech.

Saving the Generated Speech to a File

- The binary content received in response to the API request, which constitutes the generated speech in MP3 format, is written to a file named `snoop_dogg.mp3`. This operation is performed within a context manager, ensuring proper handling of the file resource.
- The action of writing to the file is facilitated by opening the file in write-binary mode (`"wb"`) and using the `write` method to save the response content.

Playing the Audio File

- The path to the saved MP3 file is stored in the variable `audio_file`.
- An external command (`mpg321` followed by the path to the MP3 file) is executed using the `os.system` method. This command invokes an MP3 player available on the system to play the audio file.
- The use of `mpg321` indicates that this script is intended to run on environments where the `mpg321` audio player is installed and accessible. Compatibility with the operating system and availability of the `mpg321` utility are assumed.

This script showcases a streamlined approach to converting text to speech using an API and playing the resulting audio, demonstrating the integration of web services and local resources in a Python application. The steps are executed sequentially, from sending the conversion request to storing and playing the audio, illustrating a practical application of text-to-speech technology.

Web Interface

This Python script exemplifies the integration of cutting-edge language modeling and text-to-speech technologies into a web application using Flask, a lightweight web framework.

At the core of this application:

Python Flask Application - Backend (app.py)

- Flask Setup: Initializes a Flask web server, providing a foundation for web-based interaction. Flask's simplicity and flexibility make it an ideal choice for deploying machine learning models in a web environment.

- Model Loading: The `transformers` pipeline for text generation is initialized with a specific model tailored to mimic Snoop Dogg's lyrical style. This showcases the power of transfer learning and natural language generation capabilities.

- Web Endpoints:
  - A root endpoint (`'/'`) that serves an HTML form, allowing users to input prompts for text generation.
  - A `'generate'` endpoint that processes form submissions, uses the loaded model to generate text based on the provided prompt, and converts this text to speech reflecting Snoop Dogg's voice, illustrating end-to-end processing from user input to audio output.

- Text-to-Speech Conversion: After generating text, the script communicates with a text-to-speech service through a POST request, sending the generated text and receiving an audio file in return. This step exemplifies how external APIs can be integrated into Python applications to extend their functionality.

- Audio Streaming: Instead of merely returning an audio file for download, the application streams the audio back to the client, enabling immediate playback in the web browser. This approach enhances user experience by providing seamless access to the generated content.

- Execution and Deployment: Instructions for running the application are straightforward, indicating the script's readiness for local development and testing. The inclusion of parameters like `debug=True` and `use_reloader=False` in the Flask app's run configuration suggests a focus on development efficiency and stability.

This script represents a confluence of natural language processing, web development, and multimedia content delivery, demonstrating how modern Python libraries and web technologies can be combined to create novel applications. It embodies the practical application of machine learning models within interactive web environments, catering to users' desire for engaging digital experiences.

```
from flask import Flask, request, render_template, send_file, Response
```

```
from transformers import pipeline
import requests

app = Flask(__name__)

# Load your model
generator = pipeline('text-generation', model="snoop-dogg-model", tokenizer='gpt2')

@app.route('/', methods=['GET'])
def home():
    # Render a simple form for input
    return render_template('index.html')

@app.route('/generate', methods=['POST'])
def generate():
    prompt = request.form['prompt']
    texts = generator(prompt, max_length=100, num_return_sequences=1)

    generated_text = texts[0]['generated_text']

    # Convert generated text to speech
    response = requests.request(
        method="POST",
        url="https://api.neets.ai/v1/tts",
        headers={
            "Content-Type": "application/json",
            "X-API-Key": "c0d9aeeef6f14e98ad46c8d23e24535b"
        },
        json={
            "text": generated_text,
            "voice_id": "snoop-dogg",
            "params": {
                "model": "ar-diff-50k"
            }
        }
    )

    with open("snoop.mp3", "wb") as f:
      f.write(response.content)

    audio_file = "snoop.mp3"

    def generate():
```

```
        with open(audio_file, "rb") as fmp3:
            data = fmp3.read(1024)
            while data:
                yield data
                data = fmp3.read(1024)

    return Response(generate(), mimetype="audio/mpeg")

if __name__ == '__main__':
    app.run(debug=True, use_reloader=False)
```

HTML Frontend (templates/index.html)

This HTML file should be placed in a folder named templates within the project directory.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Snoop Dogg Text to Speech</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 0;
            display: flex;
            justify-content: center;
            align-items: center;
            flex-direction: column;
            height: 100vh;
            background-color: #f0f0f0;
        }
        .header {
            margin-bottom: 20px;
            text-align: center;
        }
        form {
            background: white;
            padding: 20px;
            border-radius: 10px;
            box-shadow: 0 2px 4px rgba(0,0,0,0.1);
        }
```

```
        textarea {
            width: 300px;
            margin-bottom: 10px;
            padding: 10px;
            border: 1px solid #ccc;
            border-radius: 5px;
            resize: vertical;
        }
        input[type="submit"] {
            background-color: #4CAF50;
            color: white;
            padding: 10px 20px;
            border: none;
            border-radius: 5px;
            cursor: pointer;
        }
        input[type="submit"]:hover {
            background-color: #45a049;
        }
        audio {
            margin-top: 20px;
        }
    </style>
</head>
<body>
    <div class="header">
        <h1>Snoop Dogg Persona created by Mehrshad Kafi</h1>
    </div>

    <!-- Form for submitting the prompt -->
    <form id="promptForm">
        <textarea id="promptText" name="prompt" rows="4" placeholder="Enter your prompt
here..."></textarea><br>
        <input type="submit" value="Generate Speech">
    </form>

    <!-- Audio player for playing the response -->
    <audio controls id="audioPlayer">
        Your browser does not support the audio element.
    </audio>

    <script>
        document.getElementById('promptForm').addEventListener('submit',
```

```
function(event) {
            event.preventDefault(); // Prevent the default form submission
            const promptText = document.getElementById('promptText').value;

            // Use fetch API to submit the text to the Flask backend
            fetch('/generate', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded',
                },
                body: 'prompt=' + encodeURIComponent(promptText)
            })
            .then(response => response.blob()) // Convert the response to a blob
            .then(blob => {
                const url = URL.createObjectURL(blob);
                var audioPlayer = document.getElementById('audioPlayer');
                audioPlayer.src = url;
                audioPlayer.play();
            })
            .catch(error => console.error('Error:', error));
        });
    </script>
</body>
</html>
```

Ethical Considerations

In the development of our language model, particular attention was paid to the legal and ethical implications surrounding the use of data and technology. To ensure compliance with copyright laws and respect for intellectual property rights, we opted to utilize open datasets provided by Hugging Face. This platform is renowned for its extensive repository of datasets that are freely available for research and development purposes.

Furthering we incorporated a text-to-speech (TTS) system provided by Neets.ai. The selection of Neets.ai's TTS system was influenced by its open-sourced nature, ensuring that our use of voice synthesis technology aligns with principles of transparency and respect for individual privacy.