````
 load.py

```
import sys
import os
import configparser
from dotenv import load_dotenv
import logging
import tkinter as tk
from tkinter import ttk, scrolledtext
import datetime
import glob
import torch
from transformers import BartForConditionalGeneration, BartTokenizer
from transformers import GPT2LMHeadModel, GPT2Tokenizer, pipeline
import warnings
from pathlib import Path
import re
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import PyPDF2
from docx import Document
from pptx import Presentation
import pandas as pd
from transformers import AutoTokenizer, AutoModel
import numpy as np

# Ensure NLTK resources are available
import nltk
try:
    nltk.data.find('corpora/stopwords')
    nltk.data.find('tokenizers/punkt')
except LookupError:
    print("Downloading necessary NLTK resources...")
    nltk.download('stopwords')
    nltk.download('punkt')

print("Starting Yaghi's Local GPT-2 Medium LLM...")

# Set up environment variable for CUDA allocation
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'
torch.cuda.empty_cache()

# Configure logging
home_dir = str(Path.home())
log_dir = r"C:\Local_LLM\GPT2-Medium\Logs"
os.makedirs(log_dir, exist_ok=True)

# log file's path
current_datetime = datetime.datetime.now()
date_str = current_datetime.strftime("%Y%m%d")
log_filename = os.path.join(log_dir, f'log_{date_str}.log')
```
```

```
# Set up logging
logging.basicConfig(
    filename=log_filename,
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logging.info("Logging setup complete. Starting the application.")


# Load environment variables and configuration
load_dotenv()
config = configparser.ConfigParser()
config.read('config.ini')
dataset_path = "C:\\Local_LLM\\yaghiDataSet"
context_size = config.getint('DEFAULT', 'context_size', fallback=1000)
fine_tuned_model_path = config.get('DEFAULT', 'fine_tuned_model_path',
fallback="gpt2-medium")

# Ignore warnings
warnings.filterwarnings("ignore")
logging.info("Warnings are being ignored.")

# Load fine-tuned GPT-2 Medium model
try:
    logging.info(f"Attempting to load model from:
{fine_tuned_model_path}")
    logging.info(f"Successfully loaded GPT-2 Medium model")
except Exception as e:
    logging.error(f"Error loading models: {str(e)}")
    print(f"Error loading models: {str(e)}")
    sys.exit(1)

class QASystem:
    def __init__(self, dataset_path, context_size=512):
        self.dataset_path = dataset_path
        self.context_size = context_size
        self.vectorizer = TfidfVectorizer(stop_words='english')
        self.dataset, self.file_paragraph_map =
self.load_dataset(dataset_path)  # Load dataset and map paragraphs to
files
        self.paragraphs = self.split_into_paragraphs(self.dataset)  #
Split into paragraphs
        self.tfidf_matrix =
self.vectorizer.fit_transform(self.paragraphs)

        # Load GPT-2 model and tokenizer
        try:
            logging.info(f"Attempting to load model from:
{fine_tuned_model_path}")

            self.model =
GPT2LMHeadModel.from_pretrained(fine_tuned_model_path)
```

```
            self.tokenizer =
GPT2Tokenizer.from_pretrained(fine_tuned_model_path)

            logging.info(f"Successfully loaded GPT-2 Medium model")
        except Exception as e:
            logging.error(f"Error loading models: {str(e)}")
            print(f"Error loading models: {str(e)}")
            sys.exit(1)

        # Initialize the summarization model
        self.summarizer_tokenizer =
BartTokenizer.from_pretrained('facebook/bart-large-cnn')
        self.summarizer_model =
BartForConditionalGeneration.from_pretrained('facebook/bart-large-cnn')

        # Initialize the NER model for detecting names
        self.ner_model = pipeline('ner', grouped_entities=True)

    def split_into_paragraphs(self, text):
        """Splits the input text into paragraphs based on double line
breaks."""
        return [p.strip() for p in text.split('\n\n') if p.strip()]

    def load_dataset(self, dataset_path):
        dataset = ""
        file_paragraph_map = {}  # Track which paragraphs come from which
files
        for root, dirs, files in os.walk(dataset_path):
            for file in files:
                file_path = os.path.join(root, file)
                try:
                    if file.endswith('.pdf'):
                        text = self.read_pdf(file_path)
                    elif file.endswith('.docx'):
                        text = self.read_docx(file_path)
                    elif file.endswith('.pptx'):
                        text = self.read_pptx(file_path)
                    elif file.endswith('.xlsx'):
                        text = self.read_xlsx(file_path)
                    elif file.endswith('.txt'):
                        with open(file_path, 'r', encoding='utf-8') as f:
                            text = f.read()

                    # Add paragraphs to dataset and track the file they
came from
                    paragraphs = self.split_into_paragraphs(text)
                    for paragraph in paragraphs:
                        dataset += paragraph + '\n\n'
                        file_paragraph_map[paragraph] = file

                except Exception as e:
                    logging.error(f"Error reading file {file_path}:
{str(e)}")
```

```
        logging.info(f"Dataset loaded successfully. Size: {len(dataset)}
characters")
        return dataset, file_paragraph_map

    def get_answer(self, question):
        logging.info(f"Received question: {question}")
        preprocessed_question = self.preprocess_text(question)
        logging.info(f"Preprocessed question: {preprocessed_question}")

        # Extract named entities from the question
        entities_in_question = self.extract_named_entities(question)
        logging.info(f"Named entities in question:
{entities_in_question}")

        question_vector =
self.vectorizer.transform([preprocessed_question])
        similarities = cosine_similarity(question_vector,
self.tfidf_matrix)[0]

        logging.info(f"Similarities: {similarities}")

        if max(similarities) < 0.01:
            return "I apologize, but I couldn't find a relevant answer in
my knowledge base. Could you please rephrase your question or ask about a
different topic?"

        top_paragraph_indices = similarities.argsort()[-3:][::-1]
        relevant_paragraphs = [self.paragraphs[i] for i in
top_paragraph_indices]

        # Check if any of the relevant paragraphs contain the specific
named entities from the question
        paragraphs_with_entities = [p for p in relevant_paragraphs if
self.contains_named_entity(p, entities_in_question)]

        if not paragraphs_with_entities:
            return "Sorry, no information is available for the specified
individual."

        combined_text = ' '.join(paragraphs_with_entities)
        sentences = nltk.sent_tokenize(combined_text)
        sentence_similarities = cosine_similarity(question_vector,
self.vectorizer.transform(sentences))[0]

        top_sentence_indices = sentence_similarities.argsort()[-5:][::-1]
        summary = ' '.join([sentences[i] for i in top_sentence_indices])

        logging.info(f"Generated summary: {summary}")
        return summary

    def follow_up(self, question):
        """Retrieve additional relevant information and summarize."""
        logging.info(f"Follow-up for question: {question}")
        preprocessed_question = self.preprocess_text(question)
```

```
        # Extract named entities from the question
        entities_in_question = self.extract_named_entities(question)
        logging.info(f"Named entities in follow-up question:
{entities_in_question}")

        question_vector =
self.vectorizer.transform([preprocessed_question])
        similarities = cosine_similarity(question_vector,
self.tfidf_matrix)[0]

        top_paragraph_indices = similarities.argsort()[-6:][::-1]  #
Fetch more paragraphs for follow-up
        relevant_paragraphs = [self.paragraphs[i] for i in
top_paragraph_indices]

        # Filter paragraphs by named entities
        paragraphs_with_entities = [p for p in relevant_paragraphs if
self.contains_named_entity(p, entities_in_question)]

        if not paragraphs_with_entities:
            return "Sorry, no additional information is available for the
specified individual.", []

        combined_text = ' '.join(paragraphs_with_entities)
        sentences = nltk.sent_tokenize(combined_text)
        sentence_similarities = cosine_similarity(question_vector,
self.vectorizer.transform(sentences))[0]

        top_sentence_indices = sentence_similarities.argsort()[-7:][::-1]
        summary = ' '.join([sentences[i] for i in top_sentence_indices])

        # List of documents from which information was extracted
        document_list = [self.file_paragraph_map[self.paragraphs[i]] for
i in top_paragraph_indices if
self.contains_named_entity(self.paragraphs[i], entities_in_question)]

        logging.info(f"Generated follow-up summary: {summary}")
        return summary, document_list

    def preprocess_text(self, text):
        # Convert to lowercase and remove punctuation
        text = re.sub(r'[^\w\s]', '', text.lower())
        # Remove stopwords
        stop_words = set(stopwords.words('english'))
        return ' '.join([word for word in text.split() if word not in
stop_words])

    def extract_named_entities(self, text):
        """Extract named entities (like people) from the text using the
NER model."""
        entities = self.ner_model(text)
        return [entity['word'] for entity in entities if
entity['entity_group'] == 'PER']
```

```
    def contains_named_entity(self, paragraph, entities):
        """Check if a paragraph contains any of the named entities from
the question."""
        paragraph_entities = self.extract_named_entities(paragraph)
        return any(entity in paragraph_entities for entity in entities)

class QAApp:
    def __init__(self, root, qa_system):
        self.root = root
        self.qa_system = qa_system
        self.root.title("Yaghi's Local LLM")
        self.root.geometry("800x600")
        self.current_question = ""  # Store the current question being
asked
        self.last_answered_question = ""  # Store the last question that
was answered
        self.question_answered = False  # Flag to track whether a
question was answered
        self.create_widgets()

    def create_widgets(self):
        # Change the title label color to brown
        self.title_label = ttk.Label(self.root, text="Yaghi's Local LLM",
font=("Helvetica", 16), foreground="brown")
        self.title_label.pack(pady=10)
        # Create and place widgets
        self.question_label = ttk.Label(self.root, text="Enter your
prompt:")
        self.question_label.pack(pady=10)

        # Replace Entry with a Text widget for multi-line input
        self.question_entry = tk.Text(self.root, height=3, width=80)
        self.question_entry.pack(pady=10)
        self.submit_button = ttk.Button(self.root, text="Ask",
command=self.submit_question)
        self.submit_button.pack(pady=10)
        self.follow_up_button = ttk.Button(self.root, text="Follow Up",
command=self.submit_follow_up)
        self.follow_up_button.pack(pady=10)
        self.answer_label = ttk.Label(self.root, text="Answer:")
        self.answer_label.pack(pady=10)
        self.answer_text = scrolledtext.ScrolledText(self.root,
wrap=tk.WORD, width=80, height=20)
        self.answer_text.pack(pady=10)

    def submit_question(self):
        """Handles the submission of the question and resetting follow-up
tracking."""
        # Get the text from the Text widget (multi-line input)
        question = self.question_entry.get("1.0", tk.END).strip()
        if not question:
            self.answer_text.delete(1.0, tk.END)
            self.answer_text.insert(tk.END, "Please enter a question.")
```

```
            return

        self.current_question = question  # Store the current question
        logging.info(f"Question submitted: {question}")

        # Reset the last answered question and follow-up flag
        self.last_answered_question = ""
        self.question_answered = False

        # Check if it's a summarization request
        if "summarize" in question.lower() or "overview" in
question.lower():
            answer = self.qa_system.summarize_documents()
        else:
            answer = self.qa_system.get_answer(question)

        # If there is no relevant answer, notify the user
        if not answer or "I apologize" in answer:
            answer = "Sorry, no information was found in the local
dataset."
        else:
            # If an answer is found, store it as the last answered
question
            self.last_answered_question = question
            self.question_answered = True

        logging.info(f"Answer generated: {answer}")
        self.answer_text.delete(1.0, tk.END)
        self.answer_text.insert(tk.END, answer)

    def submit_follow_up(self):
        """Handles follow-up logic, ensuring it applies only to the last
answered question."""
        if not self.question_answered:
            # No question was successfully answered yet, or the last
question had no answer
            self.answer_text.delete(1.0, tk.END)
            self.answer_text.insert(tk.END, "Sorry, no information was
found in the local dataset.")
            return

        logging.info(f"Follow-up requested for question:
{self.last_answered_question}")
        summary, documents =
self.qa_system.follow_up(self.last_answered_question)

        # Format the result to include the summary and document list
        follow_up_text = f"Summary:\n{summary}\n\nDocuments Used:\n" +
"\n".join(documents)

        logging.info(f"Follow-up generated: {follow_up_text}")
        self.answer_text.delete(1.0, tk.END)
        self.answer_text.insert(tk.END, follow_up_text)
```

```
if __name__ == "__main__":
    try:
        # Check if the dataset directory exists
        if not os.path.exists(dataset_path):
            logging.error(f"Dataset directory not found: {dataset_path}")
            print(f"Dataset directory not found: {dataset_path}")
            sys.exit(1)
        root = tk.Tk()
        qa_system = QASystem(dataset_path, context_size)
        app = QAApp(root, qa_system)
        root.mainloop()
    except Exception as e:
        logging.error(f"Error in main application: {str(e)}")
        print(f"An error occurred: {str(e)}")
        import traceback
        traceback.print_exc()
        sys.exit(1)
```