```
from flask import Flask, request, render_template, session, redirect, url_for, jsonify
import PyPDF2
import io
import random
import secrets
import os
import glob
import datetime
from dotenv import dotenv_values
from groq import Groq
from json import load, dump, JSONDecodeError
import pandas as pd
import sys
import traceback

#  TF-IDF scoring
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np


env_vars = dotenv_values(".env")
Username = env_vars.get("Username", "User")
Assistantname = env_vars.get("Assistantname", "CareerBot")
GroqAPIKey = env_vars.get("GroqAPIKey")

if not GroqAPIKey:
    print("Warning: GroqAPIKey environment variable not set in .env file. Groq-powered features
will not work.")
    client = None
else:
    client = Groq(api_key=GroqAPIKey)

app = Flask(__name__)
app.secret_key = secrets.token_hex(16)


KAGGLE_RESUME_DATA_ROOT = os.path.join('data', 'data')
RESUME_CSV_PATH = os.path.join('data', 'Resume', 'Resume.csv')
GLOBAL_DATA_DIR = os.path.join('data', 'data')
CHAT_LOG_FILE = os.path.join(GLOBAL_DATA_DIR, 'ChatLog.json')
CACHED_RESUMES_FILE = os.path.join(GLOBAL_DATA_DIR, 'cached_resumes.json')

ALL_HISTORICAL_RESUMES_TEXT = []
tfidf_vectorizer = None
```

```
min_historical_tfidf_sum = 0.0
max_historical_tfidf_sum = 1.0

#  Dummy User Data for Login
USERS = {
    "user@example.com": "password123"
}



def extract_text_from_pdf(pdf_file_obj):
    """
    Extracts text from a given PDF file object (BytesIO or file-like object).
    Returns the extracted text as a string.
    """
    text = ""
    try:
        reader = PyPDF2.PdfReader(pdf_file_obj)
        for page_num in range(len(reader.pages)):
            page_text = reader.pages[page_num].extract_text()
            if page_text:
                text += page_text
        return text
    except Exception as e:
        # print(f"Error extracting text from PDF: {e}") # Suppress this for cleaner output during bulk
processing
        # traceback.print_exc() # Suppress this for cleaner output during bulk processing
        return None

def pre_process_resumes_to_cache():
    """
    Extracts text from all PDFs and CSV, and saves it to a single JSON cache file.
    This function should be run once or whenever the source data changes.
    """
    print("Starting pre-processing of historical resumes...")
    extracted_texts = []

    os.makedirs(GLOBAL_DATA_DIR, exist_ok=True)


    if os.path.exists(KAGGLE_RESUME_DATA_ROOT):
        pdf_files = glob.glob(os.path.join(KAGGLE_RESUME_DATA_ROOT, '**', '*.pdf'),
recursive=True)
        print(f"Found {len(pdf_files)} PDF files in categorized folders.")
        pdf_extracted_count = 0
        for i, pdf_path in enumerate(pdf_files):
            if (i + 1) % 100 == 0 or (i + 1) == len(pdf_files): # Print progress every 100 files or at the
end
                print(f"Processing PDF {i + 1}/{len(pdf_files)}...")
            try:
                with open(pdf_path, 'rb') as f:
                    pdf_file_obj = io.BytesIO(f.read())
                    resume_text = extract_text_from_pdf(pdf_file_obj)
                    if resume_text:
                        extracted_texts.append(resume_text)
                        pdf_extracted_count += 1
            except Exception as e:
                # print(f"Could not pre-process resume from PDF {pdf_path}: {e}") # Suppress for
cleaner output
                pass # Continue processing other files even if one fails
        print(f"Successfully extracted text from {pdf_extracted_count} PDFs.")
    else:
        print(f"Warning: Categorized resume data root '{KAGGLE_RESUME_DATA_ROOT}' not
found for pre-processing.")


    if os.path.exists(RESUME_CSV_PATH):
        try:
            df = pd.read_csv(RESUME_CSV_PATH)
            csv_extracted_count = 0

            if 'Resume_str' in df.columns:
                for text in df['Resume_str'].dropna().tolist():
                    extracted_texts.append(str(text))
                    csv_extracted_count += 1
                print(f"Pre-processed {csv_extracted_count} resumes from {RESUME_CSV_PATH}
(using 'Resume_str' column).")
            elif 'text' in df.columns:
                for text in df['text'].dropna().tolist():
                    extracted_texts.append(str(text))
                    csv_extracted_count += 1
                print(f"Pre-processed {csv_extracted_count} resumes from {RESUME_CSV_PATH}
(using 'text' column).")
            else:
                print(f"Warning: '{os.path.basename(RESUME_CSV_PATH)}' found but no
'Resume_str' or 'text' column found for pre-processing. Skipping CSV.")
        except Exception as e:
            print(f"Error pre-processing resumes from CSV {RESUME_CSV_PATH}: {e}")
            traceback.print_exc()
    else:
        print(f"Info: '{os.path.basename(RESUME_CSV_PATH)}' not found for pre-processing.
Skipping CSV.")

    print(f"Total resume texts extracted: {len(extracted_texts)}")


    try:
        if not extracted_texts:
            with open(CACHED_RESUMES_FILE, 'w', encoding='utf-8') as f:
                dump([], f, indent=4, ensure_ascii=False)
            print("No resume texts extracted. An empty cache file has been created.")
        else:
            with open(CACHED_RESUMES_FILE, 'w', encoding='utf-8') as f:
                dump(extracted_texts, f, indent=4, ensure_ascii=False)
            print(f"Successfully saved {len(extracted_texts)} resume texts to cache:
{CACHED_RESUMES_FILE}")
            if os.path.getsize(CACHED_RESUMES_FILE) == 0 and len(extracted_texts) > 0:
                print("CRITICAL WARNING: Cache file was written but is 0 bytes despite extracted
texts. Check disk space/permissions.")
    except Exception as e:
        print(f"Error saving cached resumes to {CACHED_RESUMES_FILE}: {e}")
        traceback.print_exc()
    print("Pre-processing complete.")


def load_historical_resumes():
    """
    Loads historical resume texts, prioritizing from cache.
    If cache is not available or empty, it performs full extraction.
    Also, initializes the TF-IDF vectorizer for scoring.
    """
    global ALL_HISTORICAL_RESUMES_TEXT, tfidf_vectorizer, min_historical_tfidf_sum,
max_historical_tfidf_sum
    ALL_HISTORICAL_RESUMES_TEXT = []


    if os.path.exists(CACHED_RESUMES_FILE):
        try:
            with open(CACHED_RESUMES_FILE, 'r', encoding='utf-8') as f:
                ALL_HISTORICAL_RESUMES_TEXT = load(f)
            print(f"Successfully loaded {len(ALL_HISTORICAL_RESUMES_TEXT)} resumes from
cache: {CACHED_RESUMES_FILE}")
```

```
        except (JSONDecodeError, FileNotFoundError, Exception) as e:
            print(f"Error loading from cache '{CACHED_RESUMES_FILE}': {e}. Attempting full
extraction.")
            traceback.print_exc()
            ALL_HISTORICAL_RESUMES_TEXT = [] # Ensure it's empty if loading fails
    else:
        print(f"Cache file '{CACHED_RESUMES_FILE}' not found. Performing full extraction.")


    if not ALL_HISTORICAL_RESUMES_TEXT:
        print("Performing full extraction of historical resumes (this may take time)...")
        count = 0


        if os.path.exists(KAGGLE_RESUME_DATA_ROOT):
            pdf_files = glob.glob(os.path.join(KAGGLE_RESUME_DATA_ROOT, '**', '*.pdf'),
recursive=True)
            for pdf_path in pdf_files:
                try:
                    with open(pdf_path, 'rb') as f:
                        pdf_file_obj = io.BytesIO(f.read())
                        resume_text = extract_text_from_pdf(pdf_file_obj)
                        if resume_text:
                            ALL_HISTORICAL_RESUMES_TEXT.append(resume_text)
                            count += 1
                except Exception as e:
                    pass # Suppress individual PDF errors for bulk loading
            print(f"Loaded {count} PDFs from categorized folders.")
        else:
            print(f"Warning: Categorized resume data root '{KAGGLE_RESUME_DATA_ROOT}' not
found.")


        if os.path.exists(RESUME_CSV_PATH):
            try:
                df = pd.read_csv(RESUME_CSV_PATH)
                if 'Resume_str' in df.columns:
                    for text in df['Resume_str'].dropna().tolist():
                        ALL_HISTORICAL_RESUMES_TEXT.append(str(text))
                        count += 1
                    print(f"Loaded {len(df['Resume_str'].dropna())} resumes from
{RESUME_CSV_PATH} (using 'Resume_str' column).")
                elif 'text' in df.columns:
                    for text in df['text'].dropna().tolist():
```

```
                    ALL_HISTORICAL_RESUMES_TEXT.append(str(text))
                    count += 1
                print(f"Loaded {len(df['text'].dropna())} resumes from {RESUME_CSV_PATH} (using
'text' column).")
            else:
                print(f"Warning: '{os.path.basename(RESUME_CSV_PATH)}' found but no
'Resume_str' or 'text' column found. Skipping CSV.")
        except Exception as e:
            print(f"Error loading resumes from CSV {RESUME_CSV_PATH}: {e}")
            traceback.print_exc()
    else:
        print(f"Info: '{os.path.basename(RESUME_CSV_PATH)}' not found. Skipping CSV
loading.")

    if count == 0:
        print("No historical resumes loaded from any source. Score distribution will use random
data.")
    else:
        print(f"Successfully loaded a total of {count} historical resumes after full extraction.")


  if ALL_HISTORICAL_RESUMES_TEXT:
    print("Fitting TF-IDF Vectorizer and calculating min/max historical TF-IDF sums...")
    tfidf_vectorizer = TfidfVectorizer(stop_words='english', max_features=5000)
    historical_tfidf_matrix =
tfidf_vectorizer.fit_transform(ALL_HISTORICAL_RESUMES_TEXT)


    historical_tfidf_sums = historical_tfidf_matrix.sum(axis=1)

    #Calculate min and max for scaling
    if historical_tfidf_sums.size > 0:
        min_historical_tfidf_sum = np.min(historical_tfidf_sums)
        max_historical_tfidf_sum = np.max(historical_tfidf_sums)
    else:
        min_historical_tfidf_sum = 0.0
        max_historical_tfidf_sum = 1.0


    if (max_historical_tfidf_sum - min_historical_tfidf_sum) == 0:
        max_historical_tfidf_sum = min_historical_tfidf_sum + 1.0 # Add a small epsilon to avoid
division by zero
        print("Warning: Min and Max historical TF-IDF sums are identical, adjusting max for
scaling.")
```

```
        print(f"TF-IDF Vectorizer initialized. Min historical TF-IDF sum:
{min_historical_tfidf_sum:.2f}, Max historical TF-IDF sum: {max_historical_tfidf_sum:.2f}")
    else:
        print("No historical resume texts available to initialize TF-IDF Vectorizer.")
        tfidf_vectorizer = None
        min_historical_tfidf_sum = 0.0
        max_historical_tfidf_sum = 1.0


def RealtimeInformation():
    """Returns formatted current date and time."""
    current_date_time = datetime.datetime.now()
    return current_date_time.strftime("Day: %A\nDate: %d %B %Y\nTime: %H:%M:%S\n")

def AnswerModifier(Answer):
    """Removes empty lines from the answer."""
    lines = Answer.split('\n')
    non_empty_lines = [line for line in lines if line.strip()]
    modified_answer = '\n'.join(non_empty_lines)
    return modified_answer



def calculate_resume_score(resume_text):
    """
    Calculates a numerical resume score using TF-IDF similarity to historical data.
    """
    global min_historical_tfidf_sum, max_historical_tfidf_sum
    if not resume_text or tfidf_vectorizer is None:
        return 50

    try:
        new_resume_tfidf_vector = tfidf_vectorizer.transform([resume_text])
        current_resume_tfidf_sum = new_resume_tfidf_vector.sum()


        if (max_historical_tfidf_sum - min_historical_tfidf_sum) == 0:

            if current_resume_tfidf_sum == min_historical_tfidf_sum:
                score = 90
            else:
                score = 50
        else:
            normalized_sum = (current_resume_tfidf_sum - min_historical_tfidf_sum) /
(max_historical_tfidf_sum - min_historical_tfidf_sum)
            score = 50 + (normalized_sum * 50) # Scale normalized_sum (0-1) to (50-100) range


        score = max(50, min(100, int(score)))
        return score
    except Exception as e:
        print(f"Error calculating resume score with TF-IDF: {e}")
        traceback.print_exc()
        return 50


def get_score_distribution_data():
    """
    Generates data for resume score distribution graph.
    This uses scores generated from the loaded historical resumes.
    """
    all_scores = []

    for text in ALL_HISTORICAL_RESUMES_TEXT:

        if tfidf_vectorizer:
            try:
                historical_tfidf_vector = tfidf_vectorizer.transform([text])
                hist_tfidf_sum = historical_tfidf_vector.sum()

                # Apply the same linear scaling logic
                if (max_historical_tfidf_sum - min_historical_tfidf_sum) == 0:
                    score = 90 if hist_tfidf_sum == min_historical_tfidf_sum else 50
                else:
                    normalized_sum = (hist_tfidf_sum - min_historical_tfidf_sum) /
(max_historical_tfidf_sum - min_historical_tfidf_sum)
                    score = 50 + (normalized_sum * 50)

                score = max(50, min(100, int(score)))
                all_scores.append(score)
            except Exception as e:
                all_scores.append(random.randint(50, 95))
        else:
            all_scores.append(random.randint(50, 95))

    if not all_scores:
        print("No historical resumes loaded, generating random scores for distribution.")
        for _ in range(100):
            all_scores.append(random.randint(40, 99))

    bins = {
        '0-20': 0, '21-40': 0, '41-60': 0, '61-80': 0, '81-100': 0
    }

    for score in all_scores:
        if 0 <= score <= 20: bins['0-20'] += 1
        elif 21 <= score <= 40: bins['21-40'] += 1
        elif 41 <= score <= 60: bins['41-60'] += 1
        elif 61 <= score <= 80: bins['61-80'] += 1
        elif 81 <= score <= 100: bins['81-100'] += 1

    return {
        'labels': list(bins.keys()),
        'data': list(bins.values())
    }

def generate_swot(resume_text):
    """Generates a SWOT analysis from resume text using Groq."""
    if not client:
        return "Groq API not configured. Cannot generate SWOT analysis."
    if not resume_text:
        return "No resume text provided for SWOT analysis."

    system_prompt = f"""
You are an expert career advisor.
Analyze the following resume and generate a detailed SWOT Analysis:
- Strengths
- Weaknesses
- Opportunities
- Threats

Resume Text:
{resume_text}
"""
    try:
        completion = client.chat.completions.create(
            model="llama3-70b-8192",
            messages=[{"role": "system", "content": system_prompt}],
            max_tokens=2048,
            temperature=0.5,
            top_p=1,
            stream=False,
            stop=None
        )
        answer = completion.choices[0].message.content
        return AnswerModifier(answer)
    except Exception as e:
        print(f"Error generating SWOT with Groq: {e}")
        traceback.print_exc()
        return "Failed to generate SWOT analysis. Please check Groq API key, network, or Groq
API status."

def generate_career_path(resume_text):
    """Generates predicted career paths from resume text using Groq."""
    if not client:
        return "Groq API not configured. Cannot generate career paths."
    if not resume_text:
        return "No resume text provided for career path prediction."

    system_prompt = f"""
You are a top career consultant AI.
Based on the resume below, suggest:
- 3 Entry-Level Jobs
- 3 Mid-Level Career Positions
- 3 Long-Term Senior Career Goals

Resume Text:
{resume_text}
"""
    try:
        completion = client.chat.completions.create(
            model="llama3-70b-8192",
            messages=[{"role": "system", "content": system_prompt}],
            max_tokens=2048,
            temperature=0.5,
            top_p=1,
            stream=False,
            stop=None
        )
        answer = completion.choices[0].message.content
        return AnswerModifier(answer)
    except Exception as e:
        print(f"Error generating career path with Groq: {e}")
        traceback.print_exc()
        return "Failed to predict career paths. Please check Groq API key, network, or Groq API
status."

def get_groq_response(query):
    """Gets a response from Groq based on chat history and resume context."""
    if not client:
        return "Groq API not configured. Cannot provide chatbot response."
    try:
        with open(CHAT_LOG_FILE, 'r', encoding='utf-8') as f:
            messages = load(f)
    except (FileNotFoundError, JSONDecodeError):
        messages = []
    except Exception as e:
        print(f"Error loading ChatLog.json: {e}")
        traceback.print_exc()
        messages = []

    resume_text = session.get('resume_text', None)

    system_prompt = f"""
Hello, you are {Assistantname}, an advanced Career Advisor AI.
You must use the user's resume (if provided) while answering.
If resume is not available, you can answer based on general knowledge.

User Resume:
{resume_text if resume_text else "No resume uploaded yet."}

*** Only reply in English, keep answers professional and precise. ***
"""

    system_chat = [
        {"role": "system", "content": system_prompt},
        {"role": "system", "content": RealtimeInformation()}
    ]

    messages.append({"role": "user", "content": query})

    try:
        completion = client.chat.completions.create(
            model="llama3-70b-8192",
            messages=system_chat + messages,
            max_tokens=2048,
            temperature=0.7,
            top_p=1,
            stream=False,
            stop=None
        )

        answer = completion.choices[0].message.content
        answer = answer.replace("</s>", "")

        messages.append({"role": "assistant", "content": answer})

        os.makedirs(os.path.dirname(CHAT_LOG_FILE), exist_ok=True)
        with open(CHAT_LOG_FILE, 'w', encoding='utf-8') as f:
            dump(messages, f, indent=4, ensure_ascii=False)

        return AnswerModifier(answer)
    except Exception as e:
        print(f"Error getting Groq response: {e}")
        traceback.print_exc()
        return "I'm sorry, I couldn't process your request at the moment. Please try again later.
Check Groq API key or network."

# --- Routes ---
@app.route('/login', methods=['GET', 'POST'])
def login():
    """Handles user login."""
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        if email in USERS and USERS[email] == password:
            session['logged_in'] = True
            session['email'] = email
            return redirect(url_for('index'))
        else:
            return render_template('login.html', error="Invalid credentials")
    return render_template('login.html')

@app.route('/logout')
def logout():
    """Handles user logout."""
    session.pop('logged_in', None)
    session.pop('email', None)
    return render_template('logout.html')

@app.route('/', methods=['GET', 'POST'])
def index():
    """
    Main route for resume upload and analysis.
    Handles GET request for page display and POST request for resume analysis.
    """
    logged_in = session.get('logged_in', False)

    if request.method == 'POST':
        if 'file' not in request.files:
            return jsonify({"error": "No file part"}), 400
        file = request.files['file']
        if file.filename == '':
            return jsonify({"error": "No selected file"}), 400
        if file and file.filename.endswith('.pdf'):
            try:
                pdf_file_obj = io.BytesIO(file.read())
                resume_text = extract_text_from_pdf(pdf_file_obj)

                if resume_text:
                    session['resume_text'] = resume_text

                    score = calculate_resume_score(resume_text)
                    swot = generate_swot(resume_text)
                    career_path = generate_career_path(resume_text)
                    score_distribution_data = get_score_distribution_data()

                    return jsonify({
                        "score": score,
                        "swot": swot,
                        "career_path": career_path,
                        "score_distribution_data": score_distribution_data
                    })
                else:
                    return jsonify({"error": "Could not extract text from PDF. Please ensure it's a
readable PDF."}), 400
            except Exception as e:
                import traceback
                traceback.print_exc()
                return jsonify({"error": f"Error processing file: {str(e)}. Please check the file format or
server logs."}), 500
        else:
            return jsonify({"error": "Invalid file type. Only PDF is supported."}), 400

    return render_template('index.html', logged_in=logged_in, username=Username,
assistantname=Assistantname)
```

```
@app.route('/ask', methods=['POST'])
def ask_chatbot():
    """Handles chatbot queries."""
    query = request.form.get('query')
    if not query:
        return jsonify({"answer": "Please provide a query."}), 400

    response = get_groq_response(query)
    return jsonify({"answer": response})

if __name__ == '__main__':
    # Ensure necessary base directories exist
    os.makedirs('data', exist_ok=True)
    os.makedirs(KAGGLE_RESUME_DATA_ROOT, exist_ok=True)
    os.makedirs(os.path.dirname(RESUME_CSV_PATH), exist_ok=True)

    # Check for command-line argument to trigger pre-processing
    if '--preprocess' in sys.argv:
        pre_process_resumes_to_cache()

    # Load historical resumes and initialize TF-IDF for scoring
    load_historical_resumes()

    app.run(debug=True)
```