Coding File:

Departmental assistant module:

```
import gradio as gr
from PyPDF2 import PdfReader
from langchain.text_splitter import RecursiveCharacterTextSplitter
import os
from langchain_google_genai import GoogleGenerativeAIEmbeddings
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_community.vectorstores import FAISS
from langchain.chains.question_answering import load_qa_chain
from langchain.prompts import PromptTemplate
import google.generativeai as genai
from dotenv import load_dotenv

load_dotenv()

genai.configure(api_key="AIzaSyDuV0_YnMu0wYFLATzmmk0SdshQUQPDrhk")
os.environ["GOOGLE_API_KEY"]="AIzaSyDuV0_YnMu0wYFLATzmmk0SdshQUQPDrhk"

default_pdf_path = "/Users/vasanth/PycharmProjects/EmpowerED/Data pdf fr pjt 3.pdf"

# Initialize conversation history
conversation_history = ""

def get_pdf_text(pdf_path):
    text = ""
    pdf_reader = PdfReader(pdf_path)
    for page in pdf_reader.pages:
        text += page.extract_text()
    return text

def get_text_chunks(text):
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=10000, chunk_overlap=1000)
    chunks = text_splitter.split_text(text)
    return chunks

def get_vector_store(text_chunks):
    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")
    vector_store = FAISS.from_texts(text_chunks, embeddings)
    vector_store.save_local("faiss_index")

def get_conversation_chain():
    prompt_template = """
    Answer the question as detailed as possible from the provided context, make sure to provide all the details, if
the answer is not in
    provided context just say, "answer is not available ", don't provide the wrong answer\n\n
    Context:\n {context}?\n
    Question: \n{question}\n

    Answer:
    """

    model = ChatGoogleGenerativeAI(model="gemini-pro", temperature=0.3)
    prompt = PromptTemplate(template=prompt_template, input_variables=["context", "question"])

    chain = load_qa_chain(model, chain_type="stuff", prompt=prompt)
```

```
    return chain

def user_input(user_question, pdf_file=None):
    global conversation_history

    if pdf_file is None:
        pdf_path = default_pdf_path
    else:
        pdf_path = pdf_file.name

    pdf_text = get_pdf_text(pdf_path)
    text_chunks = get_text_chunks(pdf_text)
    get_vector_store(text_chunks)

    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")
    new_db = FAISS.load_local("faiss_index", embeddings, allow_dangerous_deserialization=True)

    docs = new_db.similarity_search(user_question)

    chain = get_conversation_chain()

    # Append the current question to the conversation history
    conversation_history += f"\nUser Question:\n{user_question}\n"

    # If there is a conversation history, prepend it to the user question
    if conversation_history:
        user_question_with_history = f"{conversation_history}"
    else:
        user_question_with_history = user_question

    response = chain(
        {"input_documents": docs, "question": user_question_with_history}, return_only_outputs=True
    )

    # Extract the answer from the response
    current_answer = response["output_text"]

    # Append the current answer to the conversation history
    conversation_history += f"\nBot Answer:\n{current_answer}\n"

    return conversation_history

def main():
    iface = gr.Interface(
        fn=user_input,
        inputs=["text"],
        outputs="text",  # Set live to False to disable automatic updates
        title="                                                                                    Departmental assistant📚📖
",
        description="Your Doubts about the department",
        theme="ParityError/Interstellar"
    )

    iface.launch(share=True)

if __name__ == "__main__":
    main()
```

Resumate Module:

```
import gradio as gr
import google.generativeai as genai
import os
import docx2txt
import PyPDF2 as pdf
from dotenv import load_dotenv

# Load environment variables from a .env file
load_dotenv()

# Set up the API key
genai.configure(api_key="AIzaSyDuV0_YnMu0wYFLATzmmk0SdshQUQPDrhk")
os.environ["GOOGLE_API_KEY"]="AIzaSyDuV0_YnMu0wYFLATzmmk0SdshQUQPDrhk"

# Set up the model configuration for text generation
generation_config = {
    "temperature": 0.4,
    "top_p": 1,
    "top_k": 32,
    "max_output_tokens": 4096,
}

# Define safety settings for content generation
safety_settings = [
    {"category": f"HARM_CATEGORY_{category}", "threshold": "BLOCK_MEDIUM_AND_ABOVE"}
    for category in ["HARASSMENT", "HATE_SPEECH", "SEXUALLY_EXPLICIT",
"DANGEROUS_CONTENT"]
]

# Create a GenerativeModel instance
llm = genai.GenerativeModel(
    model_name="gemini-pro",
    generation_config=generation_config,
    safety_settings=safety_settings,
)

# Prompt Template
input_prompt_template = """
As an experienced Applicant Tracking System (ATS) analyst,
with profound knowledge in technology, software engineering, data science,
and big data engineering, your role involves evaluating resumes against job descriptions.
Recognizing the competitive job market, provide top-notch assistance for resume improvement.
Your goal is to analyze the resume against the given job description,
assign a percentage match based on key criteria, and pinpoint missing keywords accurately.
resume:{text}
description:{job_description}
I want the response in one single string having the structure
{{"Job Description Match":"%",
"Missing Keywords":"",
"Candidate Summary":"",
"Experience":""}}
"""


def evaluate_resume(job_description, resume_text):
    # Generate content based on the input text
```

```
    output = llm.generate_content(input_prompt_template.format(text=resume_text,
job_description=job_description))

    # Parse the response to extract relevant information
    response_text = output.text

    # Extracting job description match
    job_description_match = response_text.split('"Job Description Match":"')[1].split('"')[0]

    # Extracting missing keywords
    missing_keywords = response_text.split('"Missing Keywords":"')[1].split('"')[0]

    # Extracting candidate summary
    candidate_summary = response_text.split('"Candidate Summary":"')[1].split('"')[0]

    # Extracting experience
    experience = response_text.split('"Experience":"')[1].split('"')[0]

    # Return the extracted components
    return job_description_match, missing_keywords, candidate_summary, experience


# Create Gradio interface
inputs = [
    gr.Textbox(lines=10, label="Job Description"),
    gr.File(label="Upload Your Resume")
]

outputs = [
    gr.Textbox(label="Job Description Match"),
    gr.Textbox(label="Missing Keywords"),
    gr.Textbox(label="Candidate Summary"),
    gr.Textbox(label="Experience")
]

gr.Interface(
    fn=evaluate_resume,
    inputs=inputs,
    outputs=outputs,
    title="                                                                    ResuMate
",
    theme="ParityError/Interstellar"
).launch(share=True)
```

Talent Refinery Model:

```
import gradio as gr
import google.generativeai as genai
import os
from dotenv import load_dotenv

# Load environment variables from a .env file
load_dotenv()

# Set up the API key
```

```
genai.configure(api_key="AIzaSyDuV0_YnMu0wYFLATzmmk0SdshQUQPDrhk")
os.environ["GOOGLE_API_KEY"]="AIzaSyDuV0_YnMu0wYFLATzmmk0SdshQUQPDrhk"

# Set up the model configuration for text generation
generation_config = {
    "temperature": 0.4,
    "top_p": 1,
    "top_k": 32,
    "max_output_tokens": 4096,
}

# Define safety settings for content generation
safety_settings = [
    {"category": f"HARM_CATEGORY_{category}", "threshold": "BLOCK_MEDIUM_AND_ABOVE"}
    for category in ["HARASSMENT", "HATE_SPEECH", "SEXUALLY_EXPLICIT",
"DANGEROUS_CONTENT"]
]

# Create a GenerativeModel instance
llm = genai.GenerativeModel(
    model_name="gemini-pro",
    generation_config=generation_config,
    safety_settings=safety_settings,
)

# Prompt Template
input_prompt_template = """
As an experienced recruitment expert
with profound knowledge in technology, software engineering, data science,
and big data engineering, your role involves evaluating resumes against job descriptions.
Recognizing the competitive job market, provide top-notch assistance for resume improvement.
Your goal is to evaluate the resume against the given job description and write pros, cons, and your personal
reflection,
and based on the info who seems most suitable for the job and assign a percentage match based on key criteria
and explain your choice in a few sentences.
resume:{text}
description:{job_description}
I want the response in one single string having the structure
{{"Pros":"",
"Cons":"",
"Summary":"",
"Matching Percentage":""}}
"""


def evaluate_resume(job_description, resume1_text, resume2_text):
    # Generate content based on the input text for resume 1
    output1 = llm.generate_content(input_prompt_template.format(text=resume1_text,
job_description=job_description))

    # Parse the response to extract relevant information for resume 1
    response_text1 = output1.text
    Pros_of_the_first_candidate = response_text1.split('"Pros":"')[1].split("")[0]
    Cons_of_the_first_candidate = response_text1.split('"Cons":"')[1].split("")[0]
    Summary_of_the_first_candidate = response_text1.split('"Summary":"')[1].split("")[0]
    Matching_percentage_of_the_first_candidate = response_text1.split('"Matching Percentage":"')[1].split("")[0]

    # Generate content based on the input text for resume 2
    output2 = llm.generate_content(input_prompt_template.format(text=resume2_text,
job_description=job_description))
```

```
    # Parse the response to extract relevant information for resume 2
    response_text2 = output2.text
    Pros_of_the_Second_candidate = response_text2.split('"Pros":"')[1].split('"')[0]
    Cons_of_the_Second_candidate = response_text2.split('"Cons":"')[1].split('"')[0]
    Summary_of_the_Second_candidate = response_text2.split('"Summary":"')[1].split('"')[0]
    Matching_percentage_of_the_Second_candidate= response_text2.split('"Matching
Percentage":"')[1].split('"')[0]

    # Return the extracted components for each resume separately
    return Pros_of_the_first_candidate , Cons_of_the_first_candidate, Summary_of_the_first_candidate,
Matching_percentage_of_the_first_candidate, Pros_of_the_Second_candidate, Cons_of_the_Second_candidate,
Summary_of_the_Second_candidate, Matching_percentage_of_the_Second_candidate

# Create Gradio interface
inputs = [
    gr.Textbox(lines=10, label="Job Description"),
    gr.File(label="Upload First Resume"),
    gr.File(label="Upload Second Resume")
]

outputs = [
    gr.Textbox(label="Pros of the first candidate"),
    gr.Textbox(label=" Cons of the first candidate"),
    gr.Textbox(label="Summary of the first candidate"),
    gr.Textbox(label="Matching percentage of the first candidate"),
    gr.Textbox(label="Pros of the Second candidate"),
    gr.Textbox(label="Cons of the Second candidate"),
    gr.Textbox(label="Summary of the Second candidate"),
    gr.Textbox(label="Matching percentage of the Second candidate")
]

gr.Interface(
    fn=evaluate_resume,
    inputs=inputs,
    outputs=outputs,
    title="                                                                                    Candidate evaluation system -
Enhance Your Resume                                                                                                    ",
    theme="ParityError/Interstellar"
).launch()
import gradio as gr
import google.generativeai as genai
import os
from dotenv import load_dotenv

# Load environment variables from a .env file
load_dotenv()

# Set up the API key
genai.configure(api_key="AIzaSyDuV0_YnMu0wYFLATzmmk0SdshQUQPDrhk")
os.environ["GOOGLE_API_KEY"]="AIzaSyDuV0_YnMu0wYFLATzmmk0SdshQUQPDrhk"

# Set up the model configuration for text generation
generation_config = {
    "temperature": 0.4,
    "top_p": 1,
    "top_k": 32,
    "max_output_tokens": 4096,
}
```

```
# Define safety settings for content generation
safety_settings = [
    {"category": f"HARM_CATEGORY_{category}", "threshold": "BLOCK_MEDIUM_AND_ABOVE"}
    for category in ["HARASSMENT", "HATE_SPEECH", "SEXUALLY_EXPLICIT",
"DANGEROUS_CONTENT"]
]

# Create a GenerativeModel instance
llm = genai.GenerativeModel(
    model_name="gemini-pro",
    generation_config=generation_config,
    safety_settings=safety_settings,
)

# Prompt Template
input_prompt_template = """
As an experienced recruitment expert
with profound knowledge in technology, software engineering, data science,
and big data engineering, your role involves evaluating resumes against job descriptions.
Recognizing the competitive job market, provide top-notch assistance for resume improvement.
Your goal is to evaluate the resume against the given job description and write pros, cons, and your personal
reflection,
and based on the info who seems most suitable for the job and assign a percentage match based on key criteria
and explain your choice in a few sentences.
resume:{text}
description:{job_description}
I want the response in one single string having the structure
{{"Pros":"",
"Cons":"",
"Summary":"",
"Matching Percentage":""}}
"""


def evaluate_resume(job_description, resume1_text, resume2_text):
    # Generate content based on the input text for resume 1
    output1 = llm.generate_content(input_prompt_template.format(text=resume1_text,
job_description=job_description))

    # Parse the response to extract relevant information for resume 1
    response_text1 = output1.text
    Pros_of_the_first_candidate = response_text1.split('"Pros":"')[1].split('"')[0]
    Cons_of_the_first_candidate = response_text1.split('"Cons":"')[1].split('"')[0]
    Summary_of_the_first_candidate = response_text1.split('"Summary":"')[1].split('"')[0]
    Matching_percentage_of_the_first_candidate = response_text1.split('"Matching Percentage":"')[1].split('"')[0]

    # Generate content based on the input text for resume 2
    output2 = llm.generate_content(input_prompt_template.format(text=resume2_text,
job_description=job_description))

    # Parse the response to extract relevant information for resume 2
    response_text2 = output2.text
    Pros_of_the_Second_candidate = response_text2.split('"Pros":"')[1].split('"')[0]
    Cons_of_the_Second_candidate = response_text2.split('"Cons":"')[1].split('"')[0]
    Summary_of_the_Second_candidate = response_text2.split('"Summary":"')[1].split('"')[0]
    Matching_percentage_of_the_Second_candidate= response_text2.split('"Matching
Percentage":"')[1].split('"')[0]

    # Return the extracted components for each resume separately
    return Pros_of_the_first_candidate , Cons_of_the_first_candidate, Summary_of_the_first_candidate,
```

Matching_percentage_of_the_first_candidate, Pros_of_the_Second_candidate, Cons_of_the_Second_candidate, Summary_of_the_Second_candidate, Matching_percentage_of_the_Second_candidate

```
# Create Gradio interface
inputs = [
    gr.Textbox(lines=10, label="Job Description"),
    gr.File(label="Upload First Resume"),
    gr.File(label="Upload Second Resume")
]

outputs = [
    gr.Textbox(label="Pros of the first candidate"),
    gr.Textbox(label=" Cons of the first candidate"),
    gr.Textbox(label="Summary of the first candidate"),
    gr.Textbox(label="Matching percentage of the first candidate"),
    gr.Textbox(label="Pros of the Second candidate"),
    gr.Textbox(label="Cons of the Second candidate"),
    gr.Textbox(label="Summary of the Second candidate"),
    gr.Textbox(label="Matching percentage of the Second candidate")
]

gr.Interface(
    fn=evaluate_resume,
    inputs=inputs,
    outputs=outputs,
    title="                                                    Talent Refinery Model 🧠
",
    theme="ParityError/Interstellar"
).launch(share=True)
```

Booking Module:

```
import gradio as gr
import subprocess
from datetime import datetime, timedelta

# Create a dictionary to store booked slots for each AV hall
booked_slots = {'AV Hall 1': [], 'AV Hall 2': [], 'AV Hall 3': []}

# Function to check if the slot is available
def is_slot_available(av_hall, start_time, day, date):
    for slot in booked_slots[av_hall]:
        if slot['start_time'] == start_time and slot['day'] == day and slot['date'] == date:
            return False
    return True

# Function to book a slot
def book_slot(av_hall, start_time, day, date, name):
    if is_slot_available(av_hall, start_time, day, date):
        booked_slots[av_hall].append({'start_time': start_time, 'day': day, 'date': date, 'name': name})
        return True
    else:
        return False

# Function to get available slots with 50-minute duration gap
def get_available_slots(av_hall, day, date):
    available_slots = []
```

```
    current_time = datetime.strptime("08:00", "%H:%M")
    closing_time = datetime.strptime("15:15", "%H:%M")
    while current_time < closing_time:
        if is_slot_available(av_hall, current_time.strftime("%H:%M"), day, date):
            end_time = (current_time + timedelta(minutes=50)).strftime("%H:%M")
            available_slots.append(f"{current_time.strftime('%H:%M')}-{end_time}")
        current_time += timedelta(minutes=50)
    return available_slots

# Function to recommend the teacher who booked the slot last week on the same day
def recommend_teacher(av_hall, day, date):
    # Get the date of the previous week
    previous_week_date = (datetime.strptime(date, "%Y-%m-%d") - timedelta(weeks=1)).strftime("%Y-%m-
%d")
    for slot in booked_slots[av_hall]:
        if slot['day'] == day and slot['date'] == previous_week_date:
            return f"Last week on this day {slot['name']} booked."
    return None

# Function to handle booking
def av_hall_booking(av_hall, start_time, day_year, day_month, day_day, name):
    date = f"{day_year}-{day_month}-{day_day}"
    day = datetime.strptime(date, "%Y-%m-%d").strftime("%A")
    recommendation = recommend_teacher(av_hall, day, date)
    if book_slot(av_hall, start_time, day, date, name):
        if recommendation:
            success_message = f"Success! {name}, you have booked  {av_hall} for {day}, {date}, {start_time}.
{recommendation}"
        else:
            success_message = f"Success! {name}, you have booked  {av_hall} for {day}, {date}, {start_time}."
        # Speak the success message
        subprocess.run(['osascript', '-e', f'say "{success_message}"'])
        return success_message
    else:
        return f"Sorry, {name}. AV Hall {av_hall} for {day}, {date}, {start_time} is already booked."

# Gradio Interface
gr.Interface(
    fn=av_hall_booking,
    inputs=[
        gr.Dropdown(['AV Hall 1', 'AV Hall 2', 'AV Hall 3'], label="AV Hall"),
        gr.Dropdown(get_available_slots('AV Hall 1', 'Monday', '2024-03-15'), label="Start Time"),
        gr.Dropdown(['2024', '2025', '2026'], label="Year"),
        gr.Dropdown(['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12'], label="Month"),
        gr.Dropdown(['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '20',
'21', '22', '23', '24', '25', '26', '27', '28', '29', '30', '31'], label="Day"),
        gr.Textbox(label="Your Name")
    ],
    outputs=gr.Textbox(label="Booking Status"),
    title= "                                                                                                                    AV Hall  Booking
System 👩‍💻📅                                                                                                                    ",
    description="                                                                                                                    Book your slots in AV
halls                                                                                                                    ",
    theme="ParityError/Interstellar"
).launch(share=True)
```