Capstone Project

Hire Smart

Smart Resume Scanner and Analyzer

Description

This project is designed to automate the process of resume screening and matching with job descriptions. It extracts key information from resumes and job descriptions, scores candidates based on their skills, experience, and qualifications, and provides actionable insights for recruiters and job seekers.

### ****Key Features****:

* **Resume Parsing**: Extracts text from PDF and DOCX files.
* **Job Description Analysis**: Identifies required skills, role, experience, and qualifications.
* **Scoring Mechanism**: Scores candidates based on skill matching, experience, and qualifications.
* **Skill Gap Analysis**: Identifies unmatched skills and suggests relevant courses for improvement.
* **User Authentication**: Allows users to sign up, log in, and manage their profiles.
* **Real-Time Processing**: Processes resumes and job descriptions in real-time.

Features

1. **Resume Upload and Parsing**:
   * Users can upload resumes in PDF or DOCX formats.
   * Extracts key details such as name, email, phone, skills, degrees, and experience.
2. **Job Description Analysis**:
   * Extracts required skills, role, experience, and qualifications from job descriptions.
3. **Scoring and Ranking**:
   * Scores candidates based on:
     + **Skills**: Matches candidate skills with job requirements.
     + **Experience**: Evaluates candidate experience against job requirements.
     + **Qualifications**: Checks candidate degrees and marks against job requirements.
   * Ranks candidates based on their total score.
4. **Skill Gap Analysis**:
   * Identifies skills missing in the candidate’s resume compared to the job description.
   * Suggests relevant courses to bridge the skill gap.
5. **User Authentication**:
   * Allows users to sign up and log in.
   * Stores user data in a SQLite database.
6. **Real-Time Processing**:
   * Processes resumes and job descriptions in real-time.
   * Displays results immediately after processing.

Implementation

### ****Data Structures Used****:

* **User Data**: Stored in a SQLite database using Flask-SQLAlchemy. Each user has a unique ID, name, email, and password.
* **Resume Data**: Extracted using regex and stored in a dictionary for processing.
* **Job Description Data**: Extracted using regex and stored in a dictionary for comparison.

### ****Modules****:

1. **Flask Application (app.py)**:
   * Handles user authentication, file uploads, and resume processing.
   * Routes:
     + /: Home page.
     + /user\_resume: User resume upload and analysis.
     + /recruiter: Recruiter dashboard for bulk resume processing.
     + /signup and /login: User authentication.
2. **Resume Data Extraction (Resume\_extract.py)**:
   * Extracts name, email, phone, skills, degrees, marks, and experience from resumes using regex.
3. **Job Description Extraction (jobdescription\_extract.py)**:
   * Extracts required skills, role, experience, and qualifications from job descriptions using regex.
4. **Scoring Mechanism (total\_score.py)**:
   * Scores candidates based on:
     + **Skill Matching**: Compares candidate skills with job requirements.
     + **Experience**: Evaluates candidate experience against job requirements.
     + **Qualifications**: Checks candidate degrees and marks against job requirements.
5. **Text Extraction (extract.py)**:
   * Extracts text from PDF and DOCX files using pdfplumber and python-docx.
   * Handles unsupported file formats and extraction errors.
6. **Database (models.py)**:
   * Stores user data in a SQLite database using Flask-SQLAlchemy.
   * Tables:
     + User: Stores user details (name, email, password).

### ****7.Database Integration****:

* The db.py file is used to manage user data stored in the SQLite database.
* It connects to the database and retrieves user information for authentication and profile management.

### ****8.Course Recommendation Integration****:

* The recommend.py file is integrated into the resume analysis process.
* After scoring candidates, the system identifies missing skills and recommends relevant YouTube videos to bridge the skill gap.

**9.Front-End (HTML/CSS)**:

* + Responsive design using Bootstrap and custom CSS.
  + Pages:
    - home.html: Home page with options for users and recruiters.
    - user\_resume.html: Resume upload page for users.
    - results.html: Displays ranked resumes for recruiters.
    - signup.html and login.html: User authentication pages.

Code

1. **Flask Application (app.py)**:
   * Handles routing, file uploads, and resume processing.
   * Uses multithreading for concurrent resume processing.

from models import init\_db, db, User  
from flask\_paginate import Pagination, get\_page\_parameter  
from recommend import fetch\_youtube\_videos  
import os  
from flask import Flask, render\_template, request, flash, redirect, url\_for  
from werkzeug.utils import secure\_filename  
from extract import extract\_text  
from Resume\_extract import (  
 extract\_name, extract\_emails, extract\_phone\_numbers, extract\_skills,  
 extract\_bachelor\_degrees, extract\_masters\_degrees,  
 extract\_marks, extract\_12th\_qualification, extract\_10th\_qualification,  
 get\_total\_experience\_from\_resume  
)  
from Jobdescription\_extract import extract\_skills\_role\_experience\_qualification  
from total\_score import Skill,Experience,Qualification  
from concurrent.futures import ThreadPoolExecutor  
  
app = Flask(\_\_name\_\_)  
app.secret\_key = 'neha'# Upload configurations  
app.config['UPLOAD\_FOLDER'] = 'uploads'  
app.config['ALLOWED\_EXTENSIONS'] = {'pdf', 'docx'}  
MAX\_FILE\_COUNT = 100000 # Maximum number of resume files allowed per upload  
app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///data.db'  
app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False  
os.makedirs(app.config['UPLOAD\_FOLDER'], exist\_ok=True)  
db.init\_app(app)  
  
  
  
# Helper function to check valid file extensions  
def allowed\_file(filename):  
 return '.' in filename and filename.rsplit('.', 1)[1].lower() in app.config['ALLOWED\_EXTENSIONS']  
  
@app.route('/')  
def home():  
 # Query all users from the database and pass them to the template  
 users = User.query.all()  
 return render\_template('home.html')  
  
# Process resume file  
def process\_resume(resume\_file, job\_skills, job\_role, job\_experience, job\_qualification):  
 filename = secure\_filename(resume\_file.filename)  
 resume\_path = os.path.join(app.config['UPLOAD\_FOLDER'], filename)  
 resume\_file.save(resume\_path)  
  
 try:  
 resume\_text = extract\_text(resume\_path)  
 except Exception as e:  
 print(f"Error extracting text from resume: {e}")  
 return []  
  
 # Extract data from the resume  
 name = extract\_name(resume\_text)  
 emails = extract\_emails(resume\_text)  
 phone\_numbers =extract\_phone\_numbers(resume\_text)  
 skills = extract\_skills(resume\_text)  
 bachelor\_degree = extract\_bachelor\_degrees(resume\_text)  
 masters\_degree = extract\_masters\_degrees(resume\_text)  
 marks = extract\_marks(resume\_text)  
 \_12th = extract\_12th\_qualification(resume\_text)  
 \_10th = extract\_10th\_qualification(resume\_text)  
 year= get\_total\_experience\_from\_resume(resume\_text)  
  
 # Extract data from the resume  
 # Calculate the total score  
 skill\_s=Skill(skills, job\_skills)  
 experience\_s=Experience(job\_experience, year)  
 qualification\_s=Qualification(bachelor\_degree, masters\_degree, marks,job\_qualification)  
 if skill\_s.skill\_score() >0.0:  
 exp=experience\_s.experience\_score()  
 else:  
 exp=0  
  
  
 total\_score = (  
 (skill\_s.skill\_score() \* 0.50) +  
 (exp \* 0.30) +  
 (qualification\_s.qualification\_score() \* 0.20)  
 )  
  
 return [{  
 'name': name.title(),  
 'emails': emails[0],  
 'phone\_numbers': phone\_numbers[0],  
 'skills': skills,  
 'bachelor\_degree': bachelor\_degree,  
 'marks': marks,  
 '12th': \_12th,  
 '10th': \_10th,  
 'masters\_degree': masters\_degree,  
 'total\_experience':year,  
 'job\_skills': job\_skills,  
 'job\_role': job\_role,  
 'job\_experience': job\_experience,  
 'job\_qualification': job\_qualification,  
 'total\_score': total\_score  
 }]  
  
@app.route('/user\_resume', methods=['GET', 'POST'])  
def user\_resume():  
 if request.method == 'POST':  
 resume\_file = request.files.get('resume')  
 job\_description\_file = request.files.get('job\_description')  
  
 # Check if the files are uploaded  
 if not resume\_file or not job\_description\_file:  
 flash("Please upload both resume and job description files.")  
 return redirect(request.url)  
  
 # Process the job description file  
 job\_description\_filename = secure\_filename(job\_description\_file.filename)  
 job\_description\_path = os.path.join(app.config['UPLOAD\_FOLDER'], job\_description\_filename)  
 job\_description\_file.save(job\_description\_path)  
  
 try:  
 job\_description\_text = extract\_text(job\_description\_path)  
 except Exception as e:  
 flash(f"Error extracting text from job description: {e}")  
 return redirect(request.url)  
  
 # Extract job details from the job description  
 job\_skills, job\_role, job\_experience, job\_qualification = extract\_skills\_role\_experience\_qualification(job\_description\_text)  
  
 # Process the resume file  
 result = process\_resume(resume\_file, job\_skills, job\_role, job\_experience, job\_qualification)  
  
 # Extract unmatched skills  
 unmatched\_skills = [skill for skill in job\_skills if skill.lower() not in [s.lower() for s in result[0]['skills']]]  
  
 # Fetch YouTube course recommendations for unmatched skills  
 suggested\_courses = {}  
 for skill in unmatched\_skills:  
 suggested\_courses[skill] = fetch\_youtube\_videos(skill)  
  
 # Pass the data to the template  
 return render\_template('user\_results.html', result=result[0], suggested\_courses=suggested\_courses)  
  
 return render\_template('user\_resume.html')  
  
@app.route('/recruiter', methods=['GET', 'POST'])  
def recruiter():  
 if request.method == 'POST':  
 if 'resume' not in request.files or 'job\_description' not in request.files:  
 flash("No files part")  
 return redirect(request.url)  
  
 resume\_files = request.files.getlist('resume')  
 job\_description\_file = request.files.get('job\_description')  
  
 # Check if a job description file is uploaded  
 if not job\_description\_file or not allowed\_file(job\_description\_file.filename):  
 flash("Please upload a valid job description file (PDF or DOCX).")  
 return redirect(request.url)  
  
 # Check if the number of uploaded files exceeds the limit  
 if len(resume\_files) > MAX\_FILE\_COUNT:  
 flash(f"Please upload no more than {MAX\_FILE\_COUNT} resume files at a time.")  
 return redirect(request.url)  
  
 # Filter out files with disallowed extensions  
 resume\_files = [file for file in resume\_files if allowed\_file(file.filename)]  
  
 if not resume\_files:  
 flash("Please upload valid resume files (PDF or DOCX).")  
 return redirect(request.url)  
  
 # Process the job description file  
 job\_description\_filename = secure\_filename(job\_description\_file.filename)  
 job\_description\_path = os.path.join(app.config['UPLOAD\_FOLDER'], job\_description\_filename)  
 job\_description\_file.save(job\_description\_path)  
  
 try:  
 job\_description\_text = extract\_text(job\_description\_path)  
 except Exception as e:  
 flash(f"Error extracting text from job description: {e}")  
 return redirect(request.url)  
  
 # Extract job details from the job description  
 job\_skills, job\_role, job\_experience, job\_qualification = extract\_skills\_role\_experience\_qualification(  
 job\_description\_text)  
  
 # Process each resume file  
 results = []  
 with ThreadPoolExecutor() as executor:  
 futures = []  
  
 # Submit tasks for each resume file to be processed  
 for resume\_file in resume\_files:  
 futures.append(executor.submit(process\_resume, resume\_file, job\_skills, job\_role, job\_experience, job\_qualification))  
  
 # Collect results  
 for future in futures:  
 try:  
 results.extend(future.result())  
 except Exception as e:  
 flash(f"Error processing file: {e}")  
 continue  
  
 # Sort results by total\_score in descending order  
 sorted\_results = sorted(results, key=lambda x: x['total\_score'], reverse=True)  
  
 # Pagination setup  
 page = request.args.get(get\_page\_parameter(), type=int, default=1)  
 per\_page = 10 # Show 10 resumes per page  
 total = len(sorted\_results)  
 paginated\_results = sorted\_results[(page - 1) \* per\_page: page \* per\_page]  
 pagination = Pagination(page=page, total=total, per\_page=per\_page, css\_framework='bootstrap5')  
  
 return render\_template('results.html', results=paginated\_results, pagination=pagination)  
  
  
 return render\_template('upload\_form.html')  
@app.route('/signup', methods=['GET', 'POST'])  
def signup():  
 if request.method == 'POST':  
 fullname = request.form['fullname']  
 email = request.form['email']  
 password = request.form['password'] # Collect password from the form  
  
 # Check if email already exists  
 existing\_user = User.query.filter\_by(email=email).first()  
 if existing\_user:  
 return render\_template('signup.html', error="Email already exists")  
  
 # Add the new user to the database  
 new\_user = User(name=fullname, email=email, password=password) # Assuming User model has a password field  
 db.session.add(new\_user)  
 db.session.commit()  
 return redirect(url\_for('home'))  
 return render\_template('signup.html')  
  
  
@app.route('/login', methods=['GET', 'POST'])  
def login():  
 if request.method == 'POST':  
 username = request.form['username']  
 password = request.form['password']  
  
 # Check if the user exists in the database  
 user = User.query.filter\_by(name=username).first()  
  
 if user and user.password == password: # Assuming User model has a password field  
 return redirect(url\_for('recruiter', name=username))  
 else:  
 return render\_template('login.html', error="Invalid username or password")  
  
 return render\_template('login.html')  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 with app.app\_context():  
 init\_db() # Create tables  
 app.run(debug=True)

1. **Text Extraction (extract.py)**:
   * Extracts text from PDF and DOCX files.

import pdfplumber  
from docx import Document  
  
# Base class for text extraction  
class TextExtractor:  
 def extract\_text(self, file\_path):  
 raise NotImplementedError("Subclasses must implement this method")  
  
# Subclass for extracting text from DOCX files  
class DocxTextExtractor(TextExtractor):  
 def extract\_text(self, file\_path):  
 try:  
 doc = Document(file\_path)  
 text = "\n".join([paragraph.text for paragraph in doc.paragraphs])  
 return text.strip().lower()  
 except Exception as e:  
 print(f"Error reading DOCX file {file\_path}: {e}")  
 return ""  
  
# Subclass for extracting text from PDF files  
class PdfTextExtractor(TextExtractor):  
 def extract\_text(self, file\_path):  
 try:  
 with pdfplumber.open(file\_path) as pdf:  
 text = ""  
 for page in pdf.pages:  
 text += page.extract\_text()  
 return text.strip().lower()  
 except Exception as e:  
 print(f"Error reading PDF file {file\_path}: {e}")  
 return ""  
  
# Factory function to select the appropriate extractor based on file type  
def get\_text\_extractor(file\_path):  
 if file\_path.lower().endswith(".pdf"):  
 return PdfTextExtractor()  
 elif file\_path.lower().endswith(".docx"):  
 return DocxTextExtractor()  
 else:  
 print(f"Unsupported file format: {file\_path}")  
 return None  
  
# Main function to extract text from a file  
def extract\_text(file\_path):  
 extractor = get\_text\_extractor(file\_path)  
 if extractor:  
 return extractor.extract\_text(file\_path)  
 return ""

1. **Resume Data Extraction (Resume\_extract.py)**:
   * Extracts key details from resumes using regex.

import re  
from datetime import datetime  
from typing import List, Tuple  
def extract\_emails(text):  
 email\_pattern = r"[a-zA-Z0-9\_%+-.]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"  
 return re.findall(email\_pattern, text)  
  
  
def extract\_phone\_numbers(text):  
 phone\_pattern = r"\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}"  
 return re.findall(phone\_pattern, text)  
  
  
  
def extract\_skills(text):  
 skills\_pattern = r"\b(python|linux|microcontrollers|c++|embedded c|c|Communication Protocol|java|c\+\+|data science|machine learning|sql|r|excel|spring|html|css|javascript|go|ruby|GPIO|I2C|SPI|UART|CAN|ARM|AVR|PIC|RTOS|FreeRTOS|C/C++|I2C/SPI/UART|GDB|OpenOCD|JTAG debuggers|TCP/IP|UDP|MQTT|HTTP|html|css|javascript|typescript|react|angular|vue.js|node.js|spring|django|flask|ruby|go|rust|kotlin|swift|dart|flutter|tensorflow|pytorch|deep learning|data science|nlp|big data|hadoop|spark|kafka|docker|kubernetes|aws|azure|gcp|git|ci/cd|devops|microservices|rest api|graphql|gdb|assembly|vhdl|verilog|fpga|matlab|simulink)\b"  
 skills = re.findall(skills\_pattern, text, re.IGNORECASE)  
 skills\_found = list(set([skill.lower() for skill in skills if skill.lower() not in ['r', 'following']]))  
 return skills\_found  
  
  
def extract\_experience(text):  
 experience\_pattern = r"(\d+)\s\*(years|month[s]?)\s\*(experience|exp)"  
 return re.findall(experience\_pattern, text)  
  
  
def extract\_name(text):  
 lines = text.split('\n')  
 name = lines[0]  
 return name  
  
  
def extract\_bachelor\_degrees(text):  
 # Define the full names of the degrees  
 bachelor\_degrees = {  
 "Bachelor of Technology": ["BTech", "B.Tech", "BTech.", "B.TECH", "BTech", "b. tech ",  
 "Bachelor of Technology "],  
 "Bachelor of Engineering": ["BE", "B.E.", "B.E", "BEng", "B.Eng", "Bachelor of Engineering"],  
 "Bachelor of Science": ["BSc", "B.Sc", "BSc.", "B.Sc.", "B.Sci", "B.S.", "Bachelor of Science"],  
 "Bachelor of Arts": ["BA", "B.A.", "B.A", "BArt", "B.Ars", "Bachelor of Arts"],  
 "Bachelor of Commerce": ["BCom", "B.Com", "BCom.", "B.Com.", "Bachelor of Commerce"],  
 "Bachelor of Design": ["BDes", "B.Des", "Bachelor of Design"],  
 "Bachelor of Architecture": ["BArch", "B.Arch", "Bachelor of Architecture"],  
 "Bachelor of Education": ["BEd", "B.Ed", "B.Ed.", "Bachelor of Education"],  
 "Bachelor of Fine Arts": ["BFA", "B.F.A.", "B.F.A", "Bachelor of Fine Arts"],  
 "Bachelor of Business Administration": ["BBA", "B.B.A.", "Bachelor of Business Administration"],  
 "Bachelor of Computer Applications": ["BCA", "B.C.A.", "Bachelor of Computer Applications"]  
 }  
  
 # List to store the found degree names  
 degrees\_found = []  
  
 # Iterate through the dictionary and check if any abbreviation or full name is found  
 for full\_name, abbreviations in bachelor\_degrees.items():  
 # Regex for matching both full names and abbreviations (case-insensitive)  
 for abbr in abbreviations:  
 # Search for the degree in the text, case-insensitive  
 if re.search(r"\b" + re.escape(abbr) + r"\b", text, re.IGNORECASE):  
 degrees\_found.append(full\_name)  
 break # Once we find the full degree name, no need to check other abbreviations for this degree  
  
 # Return the list of degrees found  
 return degrees\_found  
  
  
def extract\_masters\_degrees(text):  
 # Define the full names of the degrees  
 masters\_degrees = {  
 "Master of Technology": ["MTech", "M.Tech", "MTech.", "M.TECH", "MTech", "Master of Technology"],  
 "Master of Engineering": ["ME", "M.E.", "M.E", "MEng", "M.Eng", "Master of Engineering"],  
 "Master of Science": ["MSc", "M.Sc", "MSc.", "M.Sc.", "M.Sci", "M.S.", "Master of Science"],  
 "Master of Arts": ["MA", "M.A.", "M.A", "MArt", "M.Ars", "Master of Arts"],  
 "Master of Commerce": ["MCom", "M.Com", "MCom.", "M.Com.", "Master of Commerce"],  
 "Master of Design": ["MDes", "M.Des", "Master of Design"],  
 "Master of Architecture": ["m.Arch", "M.Arch", "Master of Architecture"],  
 "Master of Education": ["MEd", "M.Ed", "M.Ed.", "Master of Education"],  
 "Master of Fine Arts": ["MFA", "M.F.A.", "M.F.A", "Master of Fine Arts"],  
 "Master of Business Administration": ["MBA", "M.B.A.", "Master of Business Administration"],  
 "Master of Computer Applications": ["MCA", "M.C.A.", "Master of Computer Applications"]  
 }  
  
 # List to store the found degree names  
 degrees\_found = []  
  
 # Iterate through the dictionary and check if any abbreviation or full name is found  
 for full\_name, abbreviations in masters\_degrees.items():  
 # Regex for matching both full names and abbreviations (case-insensitive)  
 for abbr in abbreviations:  
 # Search for the degree in the text, case-insensitive  
 if re.search(r"\b" + re.escape(abbr) + r"\b", text, re.IGNORECASE):  
 degrees\_found.append(full\_name)  
 break # Once we find the full degree name, no need to check other abbreviations for this degree  
  
 # Return the list of degrees found  
 return degrees\_found  
  
  
def extract\_marks(text):  
 # Regex patterns to match different marks formats  
 marks\_patterns = [  
 # Matches CGPA formats (e.g., CGPA 8.5, CGPA: 9.2, CGPA=7.8)  
 r"\bCGPA\s\*[:=]?\s\*(\d+(\.\d+)?)\b",  
 r"\bC\.G\.P\.A\s\*[:=]?\s\*(\d+(\.\d+)?)\b",  
 r"\bCGPA\s\*=\s\*(\d+(\.\d+)?)\b",  
  
 # Matches percentage formats (e.g., 85%, 92.5%, 75 %)  
 r"\b(\d+(\.\d+)?)\s\*%?\b(?=\s\*(?:percentage|percent|%))",  
  
 # Matches aggregate format (e.g., Aggregate: 80%, Aggregate CGPA 7.9)  
 r"\bAggregate\s\*[:=]?\s\*(\d+(\.\d+)?)\s\*(%|CGPA)?\b",  
  
 # Matches marks in general (e.g., Marks: 75/100, 90/100)  
 r"\bMarks\s\*[:=]?\s\*(\d+(\.\d+)?)/(\d+)\b",  
  
 # Matches GPA formats (e.g., GPA 3.5, GPA: 4.0)  
 r"\bGPA\s\*[:=]?\s\*(\d+(\.\d+)?)\b",  
  
 # Matches aggregate percentage (e.g., Aggregate percentage: 85%)  
 r"\bAggregate\s\*percentage\s\*[:=]?\s\*(\d+(\.\d+)?)\s\*%\b",  
  
 # Matches first-class, second-class, etc. (if present, this will be caught as a degree descriptor)  
 r"\b(First|Second|Third)\s\*Class\b",  
 ]  
  
 # List to store the found marks or grades  
 marks\_found = []  
  
 # Iterate through each regex pattern and search for matches  
 for pattern in marks\_patterns:  
 matches = re.findall(pattern, text, re.IGNORECASE)  
  
 for match in matches:  
 # Extract the mark or grade value from the match  
 if match[0]: # if the first group (value) is not empty  
 marks\_found.append(match[0])  
  
 # Return the list of marks found  
 return marks\_found  
  
  
def extract\_12th\_qualification(text):  
 # Define the full names and abbreviations for 12th qualifications  
 twelfth\_qualifications = {  
 "12th Grade": ["12th", "12th grade", "12th standard", "12th class", "12th"],  
 "Higher Secondary": ["Higher Secondary", "Higher Sec.", "H.S.", "HS", "Higher Secondary School"],  
 "Intermediate": ["Intermediate", "Inter", "Intermed.", "12th Intermediate", "Intermediate Class"],  
 "Pre-University": ["Pre-University", "PU", "PUC", "Pre-U", "Pre-U Class"],  
 "Senior Secondary": ["Senior Secondary", "Senior Sec.", "SS", "Senior Secondary School"]  
 }  
  
 # List to store the found 12th qualifications  
 qualifications\_found = []  
  
 # Iterate through the dictionary and check if any abbreviation or full name is found  
 for full\_name, abbreviations in twelfth\_qualifications.items():  
 for abbr in abbreviations:  
 if re.search(r"\b" + re.escape(abbr) + r"\b", text, re.IGNORECASE):  
 qualifications\_found.append(full\_name)  
 break # Once found, no need to check other abbreviations for this qualification  
  
 # Return the list of qualifications found  
 return qualifications\_found  
  
  
def extract\_10th\_qualification(text):  
 # Define the full names and abbreviations for 10th qualifications  
 tenth\_qualifications = {  
 "10th Grade": ["10th", "10th grade", "10th standard", "10th class", "10th"],  
 "Secondary School": ["Secondary School", "Sec. School", "Secondary Education", "Secondary", "SSC", "S.S.C.",  
 "S.S.C"],  
 "Matriculation": ["Matriculation", "Matric", "Matric Class", "10th Matric"],  
 "Senior Secondary": ["Senior Secondary", "Senior Sec.", "SS", "Senior Secondary School"]  
 }  
  
 # List to store the found 10th qualifications  
 qualifications\_found = []  
  
 # Iterate through the dictionary and check if any abbreviation or full name is found  
 for full\_name, abbreviations in tenth\_qualifications.items():  
 for abbr in abbreviations:  
 if re.search(r"\b" + re.escape(abbr) + r"\b", text, re.IGNORECASE):  
 qualifications\_found.append(full\_name)  
 break # Once found, no need to check other abbreviations for this qualification  
  
 # Return the list of qualifications found  
 return qualifications\_found  
  
  
def extract\_experience\_section(resume\_text: str) -> str:  
 *"""  
 Extracts the experience section from the resume text.  
 """* experience\_pattern = re.compile(r"(experience)(.\*?)(projects|education|skills|$)", re.DOTALL | re.IGNORECASE)  
 match = experience\_pattern.search(resume\_text)  
 if match:  
 return match.group(2).strip()  
 return ""  
  
  
def extract\_durations(resume\_text: str) -> List[str]:  
 *"""  
 Extracts all duration strings from the resume text.  
 """* duration\_pattern = re.compile(  
 r"\b(?:january|february|march|april|may|june|july|august|september|october|november|december)\s?\d{4}\s\*(?:[-–]\s\*(?:present|\b(?:january|february|march|april|may|june|july|august|september|october|november|december)\s?\d{4})?)\b",  
 re.IGNORECASE  
 )  
 durations = duration\_pattern.findall(resume\_text)  
 return [duration.strip() for duration in durations]  
  
  
def parse\_date(date\_str: str) -> datetime:  
 *"""  
 Parses a date string into a datetime object.  
 Handles both "Month Year" and "Year" formats.  
 """* try:  
 return datetime.strptime(date\_str.strip(), "%B %Y") # Try to parse as Month Year  
 except ValueError:  
 return datetime.strptime(date\_str.strip(), "%Y") # Fallback to Year if Month is missing  
  
  
def merge\_overlapping\_ranges(ranges: List[Tuple[datetime, datetime]]) -> List[Tuple[datetime, datetime]]:  
 *"""  
 Merges overlapping date ranges into a single range.  
 """* if not ranges:  
 return []  
  
 # Sort ranges by start date  
 sorted\_ranges = sorted(ranges, key=lambda x: x[0])  
  
 merged\_ranges = []  
 current\_range = sorted\_ranges[0]  
  
 for next\_range in sorted\_ranges[1:]:  
 if next\_range[0] <= current\_range[1]: # Overlapping ranges  
 current\_range = (min(current\_range[0], next\_range[0]), max(current\_range[1], next\_range[1]))  
 else:  
 merged\_ranges.append(current\_range)  
 current\_range = next\_range  
  
 merged\_ranges.append(current\_range)  
 return merged\_ranges  
  
  
def calculate\_total\_experience(durations: List[str]) -> str:  
 *"""  
 Calculates the total experience by merging overlapping ranges and summing the durations.  
 """* ranges = []  
  
 for duration in durations:  
 if '–' in duration or '-' in duration: # Ensure we're dealing with a valid range  
 start, end = re.split(r'[–-]', duration) # Split on either en dash or hyphen  
  
 # Parse start and end dates  
 start\_date = parse\_date(start.strip())  
 if "present" in end.strip().lower():  
 end\_date = datetime.now() # Use current date for present  
 else:  
 end\_date = parse\_date(end.strip())  
  
 ranges.append((start\_date, end\_date))  
  
 # Merge overlapping ranges  
 merged\_ranges = merge\_overlapping\_ranges(ranges)  
  
 # Calculate total months  
 total\_months = 0  
 for start, end in merged\_ranges:  
 delta = (end.year - start.year) \* 12 + (end.month - start.month)  
 total\_months += delta  
  
 # Convert total months to years and months  
 years = total\_months // 12  
 months = total\_months % 12  
  
 return f"{years} years and {months} months"  
  
  
def get\_total\_experience\_from\_resume(resume\_text: str) -> str:  
 *"""  
 Extracts the experience section, extracts durations, and calculates the total experience.  
 """* # Step 1: Extract the experience section  
 experience\_section = extract\_experience\_section(resume\_text)  
  
 if not experience\_section:  
 return "No experience section found."  
  
 # Step 2: Extract durations from the experience section  
 durations = extract\_durations(experience\_section)  
  
 if not durations:  
 return "No durations found in the experience section."  
  
 # Step 3: Calculate the total experience  
 total\_experience = calculate\_total\_experience(durations)  
  
 return total\_experience

1. **Job Description Extraction (jobdescription\_extract.py)**:
   * Extracts required skills, role, experience, and qualifications from job descriptions.

import re  
from abc import ABC, abstractmethod  
  
# Base abstract class for text extraction  
class TextExtractor(ABC):  
 @abstractmethod  
 def extract(self, text):  
 *"""Subclasses must implement this method"""* pass  
  
# Subclass for extracting skills  
class SkillsExtractor(TextExtractor):  
 def extract(self, text):  
 skills\_pattern = r"\b(python|linux|microcontrollers|c++|embedded c|c|communication protocol|java|c\+\+|data science|machine learning|deep learning|nlp|sql|r|excel|spring|html|css|javascript|go|ruby|react|node\.js|angular|typescript|django|flask|tensorflow|pytorch|keras|matlab|sas|power bi|tableau|git|github|docker|kubernetes|ansible|jenkins|ci/cd|aws|azure|gcp|devops|bash|shell scripting|golang|rust|swift|kotlin|objective-c|php|perl|scala|big data|hadoop|spark|nosql|mongodb|postgresql|firebase|redis|graphql|json|xml|rest api|soap|ui/ux|figma|adobe xd|flutter|dart|swiftui|xamarin|robotics|automation|gpio|i2c|spi|uart|can|arm cortex|avr|pic|rtos|freertos|c/c++|i2c/spi/uart|gdb|openocd|jtag debuggers|tcp/ip|udp|mqtt|http|blockchain|smart contracts|solidity|cybersecurity|penetration testing|ethical hacking|network security|firewalls|cryptography|cloud computing)\b"  
  
 skills = re.findall(skills\_pattern, text, re.IGNORECASE)  
 # Clean up skills by removing duplicates and irrelevant words  
 skills = list(set([skill.lower() for skill in skills if skill.lower() not in ['r', 'following']]))  
 skills.sort()  
 return skills  
  
  
# Subclass for extracting role  
class RoleExtractor(TextExtractor):  
 def extract(self, text):  
 role\_pattern = r"(role|position|title|Job Role):?\s\*([A-Za-z\s]+)(?=\s\*at|\s\*$)"  
 roles = re.findall(role\_pattern, text, re.IGNORECASE)  
 return roles[0][1] if roles else "Not found"  
  
# Subclass for extracting experience years  
class ExperienceExtractor(TextExtractor):  
 def extract(self, text):  
 years\_pattern = r"\b(\d+\s\*(?:\+|to|-)?\s\*\d\*)\s\*(?:years?|yrs?)\b"  
 experience\_years = re.findall(years\_pattern, text, re.IGNORECASE)  
 return experience\_years[0] if experience\_years else "Not found"  
  
# Subclass for extracting qualification  
class QualificationExtractor(TextExtractor):  
 def extract(self, text):  
 qualification\_pattern = r"\b(?:Undergraduate|Graduate|Post[-\s]?Graduate|Postgraduate|B(?:\.?Tech|\.?E|achelors?)|M(?:\.?Tech|\.?E|asters?)|B\.?Sc|M\.?Sc|B\.?A|M\.?A|Ph\.?D|Diploma|Associate\sDegree)\b"  
 qualifications = list(set(re.findall(qualification\_pattern, text, re.IGNORECASE)))  
 return qualifications  
  
# Main class for extracting all details  
class ResumeExtractor:  
 def \_\_init\_\_(self):  
 self.skill\_extractor = SkillsExtractor()  
 self.role\_extractor = RoleExtractor()  
 self.experience\_extractor = ExperienceExtractor()  
 self.qualification\_extractor = QualificationExtractor()  
  
 def extract\_details(self, text):  
 skills = self.skill\_extractor.extract(text)  
 role = self.role\_extractor.extract(text)  
 years\_of\_experience = self.experience\_extractor.extract(text)  
 qualification = self.qualification\_extractor.extract(text)  
 return skills, role, years\_of\_experience, qualification  
  
# Example usage  
def extract\_skills\_role\_experience\_qualification(text):  
 extractor = ResumeExtractor()  
 skills, role, years\_of\_experience, qualification = extractor.extract\_details(text)  
 return skills, role, years\_of\_experience, qualification

1. **Scoring Mechanism (total\_score.py)**:
   * Scores candidates based on skill matching, experience, and qualifications.

class Skill:  
 *"""For Calculating the skill score of each resume according to mathched skills with  
 the job description"""* def \_\_init\_\_(self, skills, job\_skills):  
 *"""skills and job skills used to compare the data"""* self.skills = skills  
 self.job\_skills = job\_skills  
  
 def skill\_score(self):  
 *"""Firstly convert each skill to lower characters and then mathched  
 according to how many skills mathched with job description : (skill\_mathched/length of job description skill)\*100"""* candidate\_skills\_normalized = {skill.lower() for skill in self.skills}  
 job\_skills\_normalized = {skill.lower() for skill in self.job\_skills}  
 # Find the intersection of both sets to get the matching skills  
 matching\_skills = candidate\_skills\_normalized & job\_skills\_normalized  
 # Count the matching skills  
 skill\_matching\_count = len(matching\_skills)  
 # skill\_score  
 skill\_score = (skill\_matching\_count / len(self.job\_skills)) \* 100  
 return skill\_score  
  
class Experience:  
 *"""For calculating the experience and match with the job description if resume experience is  
 in the range of job description then experience\_score is 100, else the experience is non-zero then 20, and  
 if the experience is between 0.5 to min job-experience then experience score is 50"""* def \_\_init\_\_(self, job\_experience, year):  
 self.job\_experience = job\_experience  
 self.year = year  
  
 def experience\_score(self):  
 try:  
 experience = float(self.year.split()[0]) # Extract the number of years  
 except ValueError:  
 experience = 0 # Default to 0 if format is unexpected  
  
 experience\_matching\_count = 0  
  
 if 'to' in self.job\_experience:  
 experience\_range = self.job\_experience.split('to')  
 start = float(experience\_range[0].strip())  
 end = float(experience\_range[1].strip())  
 if start <= experience <= end:  
 experience\_matching\_count = 1  
 elif 0.5 <= experience < start:  
 experience\_matching\_count = 0.5  
 elif experience > end:  
 experience\_matching\_count = 0.8 # Overqualified candidates get 80%  
 else:  
 experience\_matching\_count = 0  
  
 elif '+' in self.job\_experience:  
 min\_experience = float(self.job\_experience.replace('+', '').strip())  
 if experience >= min\_experience:  
 experience\_matching\_count = 1  
 elif 0.5 <= experience < min\_experience:  
 experience\_matching\_count = 0.5  
 else:  
 experience\_matching\_count = 0  
  
 elif '-' in self.job\_experience:  
 experience\_range = self.job\_experience.split('-')  
 start = float(experience\_range[0].strip())  
 end = float(experience\_range[1].strip())  
 if start <= experience <= end:  
 experience\_matching\_count = 1  
 elif 0.5 <= experience < start:  
 experience\_matching\_count = 0.5  
 elif experience > end:  
 experience\_matching\_count = 0.8 # Overqualified candidates get 80%  
 else:  
 experience\_matching\_count = 0  
  
 else:  
 experience\_required = float(self.job\_experience)  
 if experience >= experience\_required:  
 experience\_matching\_count = 1  
 else:  
 experience\_matching\_count = 0  
  
 experience\_score = experience\_matching\_count \* 100  
 return experience\_score  
  
  
class Qualification:  
 def \_\_init\_\_(self, bachelor\_degree, masters\_degree, marks, job\_qualification):  
 self.bachelor\_degree = bachelor\_degree  
 self.masters\_degree = masters\_degree  
 self.marks = marks  
 self.job\_qualification = job\_qualification  
  
 def check\_qualification(self, degree, job\_qualification):  
 # Convert qualifications to lowercase for case-insensitive comparison  
 candidate\_qualifications\_normalized = [qualification.lower() for qualification in degree]  
 required\_qualifications\_normalized = [qualification.lower() for qualification in job\_qualification]  
  
 matching\_qualifications = [qualification for qualification in candidate\_qualifications\_normalized if  
 any(req in qualification for req in required\_qualifications\_normalized)]  
 return matching\_qualifications  
  
 def marks\_converter(self, marks\_10th, marks\_12th, cgpa):  
 # Convert the marks to a consistent scale  
 if 0 <= marks\_10th <= 10:  
 marks\_10th = round(marks\_10th \* 9.5)  
 if 0 <= marks\_12th <= 10:  
 marks\_12th = round(marks\_12th \* 9.5)  
 if 0 <= cgpa <= 10:  
 cgpa = round(cgpa \* 9.5)  
 return marks\_10th, marks\_12th, cgpa  
  
 def check\_qualification\_criteria(self, marks\_10th, marks\_12th, cgpa):  
 qualification\_matching\_count = 0  
 marks\_10th, marks\_12th, cgpa = self.marks\_converter(marks\_10th, marks\_12th, cgpa)  
 if marks\_10th >= 75.00 and marks\_12th >= 65.00 and cgpa >= 65.00:  
 qualification\_matching\_count = 1  
 else:  
 qualification\_matching\_count = 0  
 return qualification\_matching\_count  
  
 def qualification\_score(self):  
 degree = self.bachelor\_degree or self.masters\_degree  
 qualification\_matching = self.check\_qualification(degree, self.job\_qualification)  
 if qualification\_matching:  
 qualification\_check = self.check\_qualification\_criteria(  
 float(self.marks[2]), float(self.marks[1]), float(self.marks[0])  
 )  
 qualification\_score = qualification\_check \* 100  
 else:  
 qualification\_score = 0  
 return qualification\_score

1. **Database (models.py)**:
   * Manages user data using Flask-SQLAlchemy.

from flask\_sqlalchemy import SQLAlchemy  
  
db = SQLAlchemy()  
  
class User(db.Model):  
 id = db.Column(db.Integer, primary\_key=True)  
 name = db.Column(db.String(80), nullable=False)  
 email = db.Column(db.String(120), unique=True, nullable=False)  
 password = db.Column(db.String(128), nullable=False)  
  
class SavedResume(db.Model):  
 id = db.Column(db.Integer, primary\_key=True)  
 recruiter\_name = db.Column(db.String(100), nullable=False) # Recruiter who saved the resume  
 candidate\_name = db.Column(db.String(100), nullable=False)  
 email = db.Column(db.String(100), nullable=True)  
 phone = db.Column(db.String(20), nullable=True)  
 skills = db.Column(db.Text, nullable=False)  
 experience = db.Column(db.String(50), nullable=True)  
 qualification = db.Column(db.String(100), nullable=True)  
 total\_score = db.Column(db.Float, nullable=False)  
  
 def \_\_init\_\_(self, recruiter\_name, candidate\_name, email, phone, skills, experience, qualification, total\_score):  
 self.recruiter\_name = recruiter\_name  
 self.candidate\_name = candidate\_name  
 self.email = email  
 self.phone = phone  
 self.skills = skills  
 self.experience = experience  
 self.qualification = qualification  
 self.total\_score = total\_score  
  
def init\_db():  
 # Creates all the tables  
 db.create\_all()

Output

Home Page

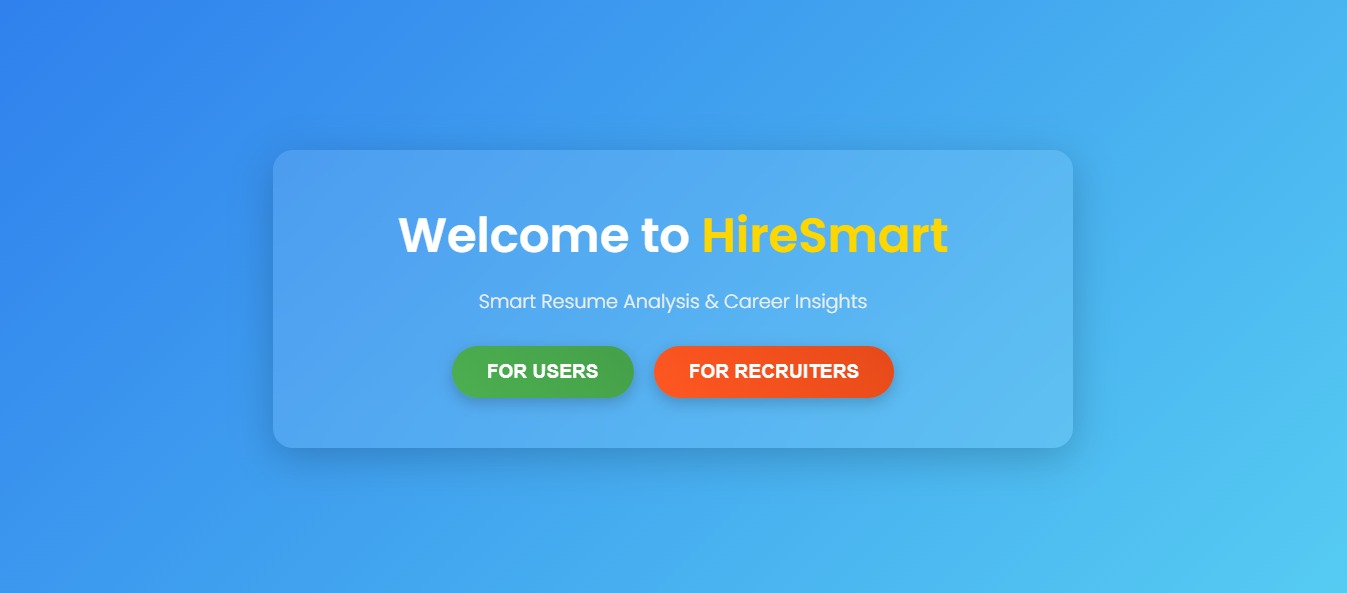


Image 1: For User Page

(Uploading resume and job description)

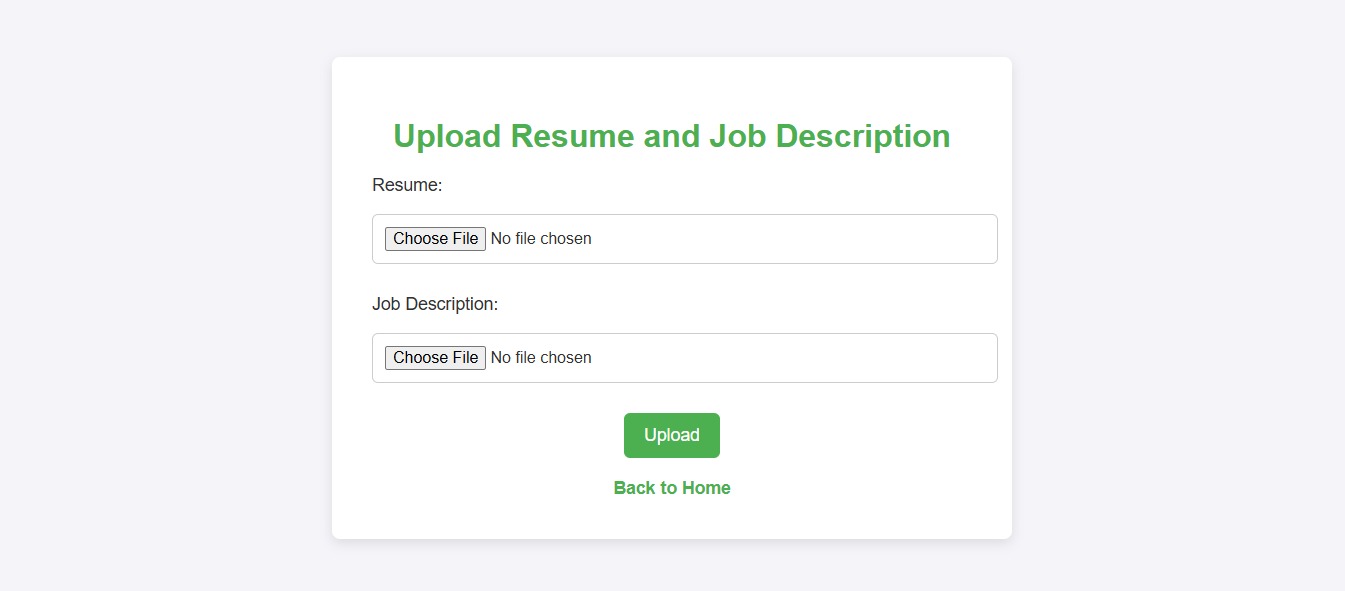


Image 2: Score and course suggestion

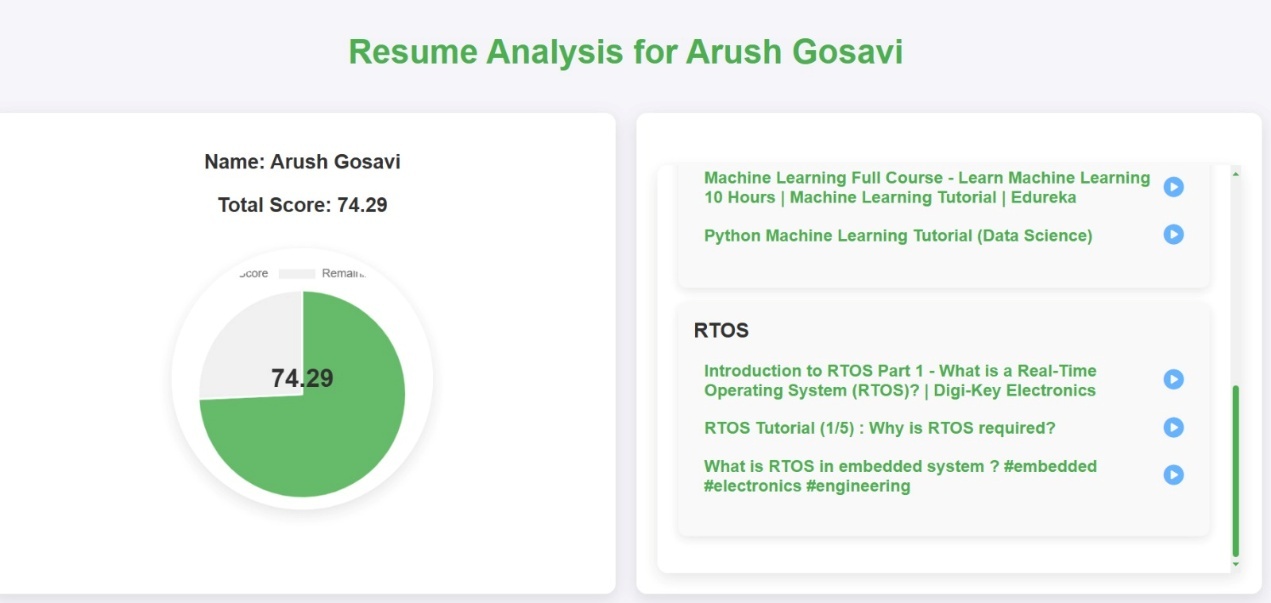


Image 3: For Recruiter page

(Signup)

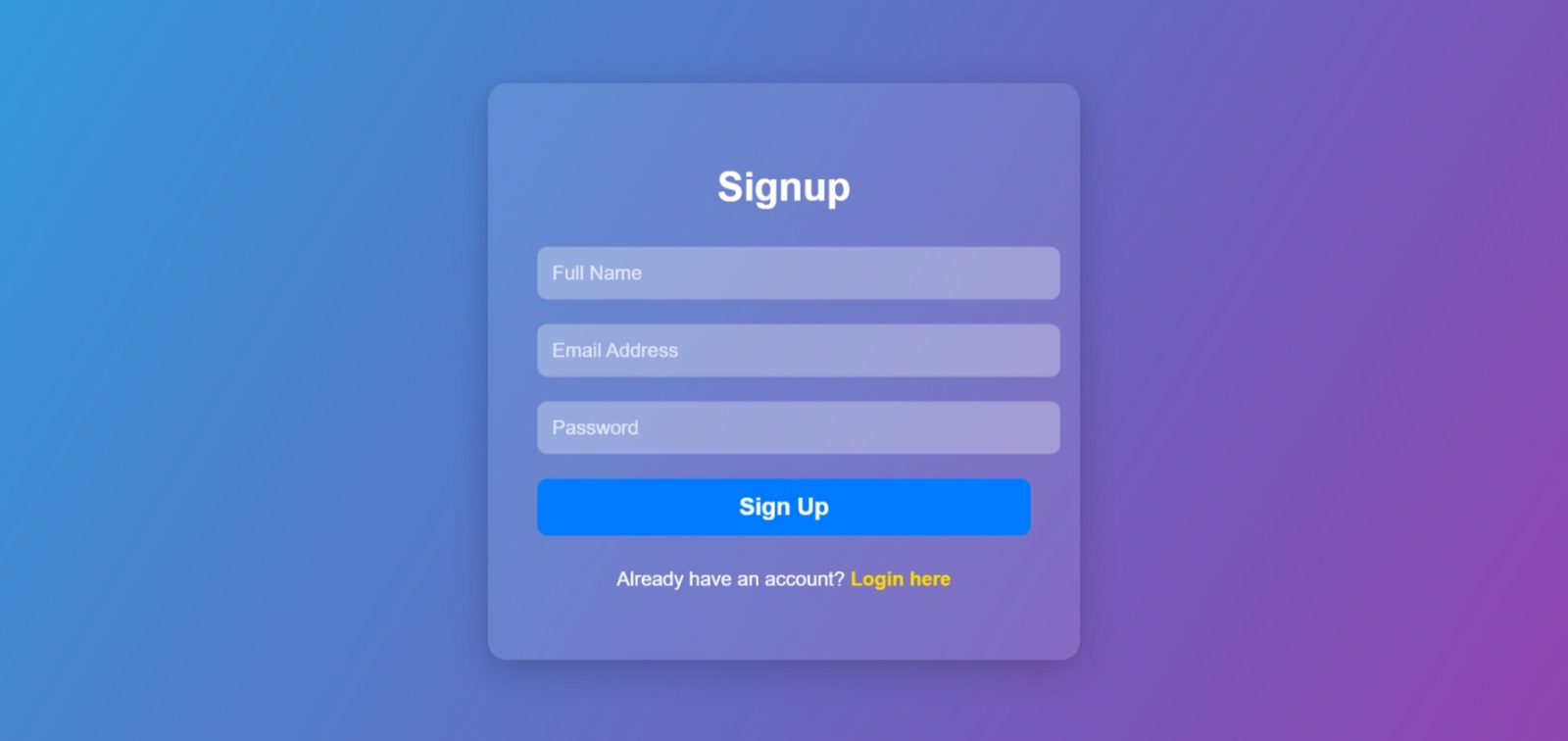


Image 4: Login Page

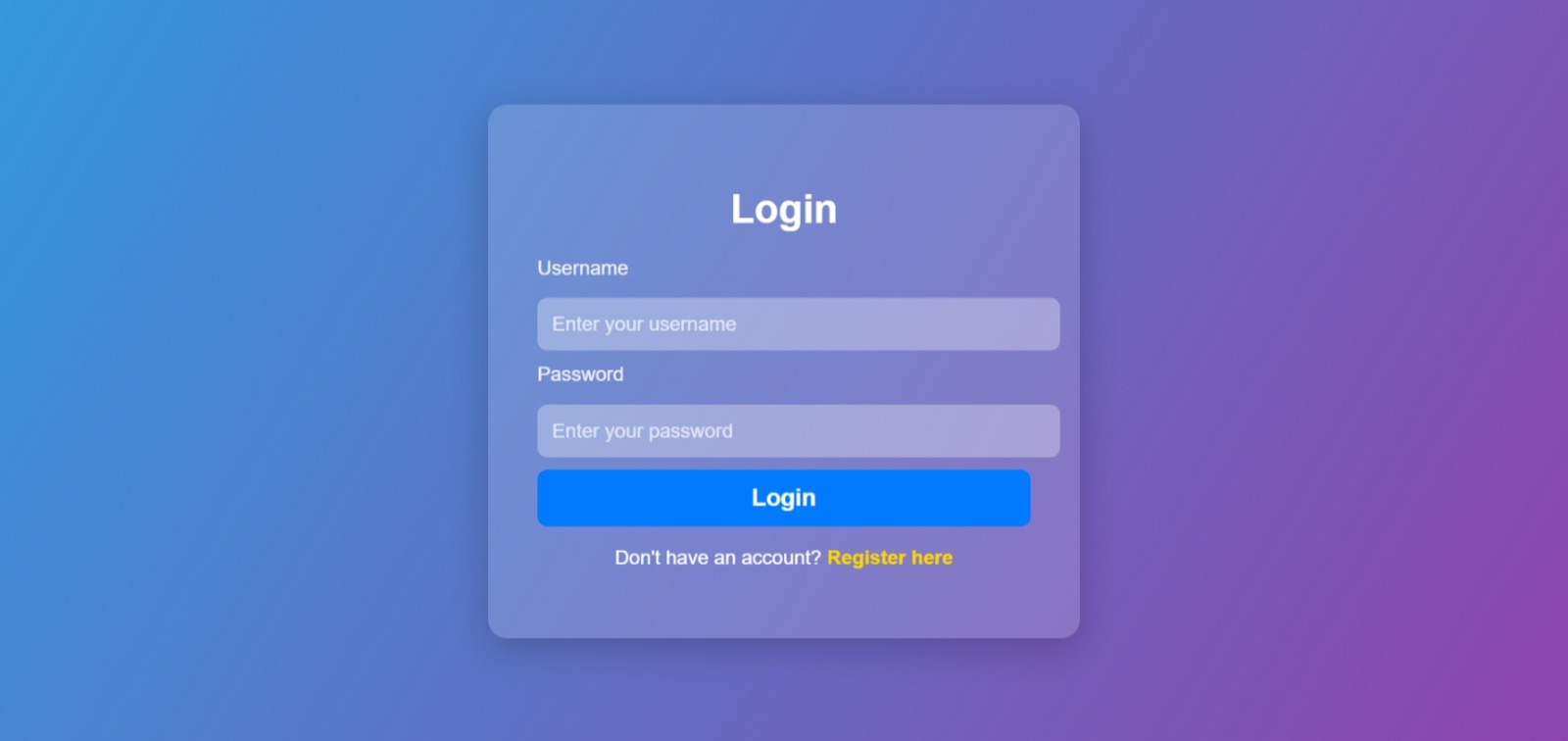


Image 5: Uploading Multiple resume and job description

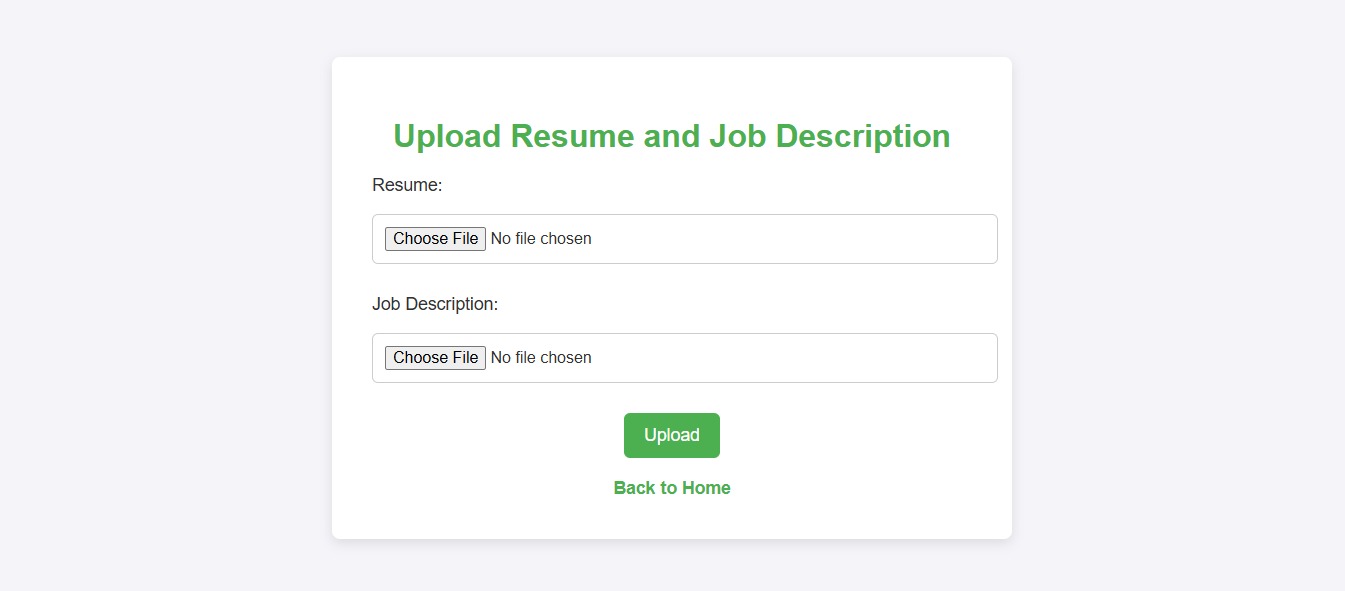
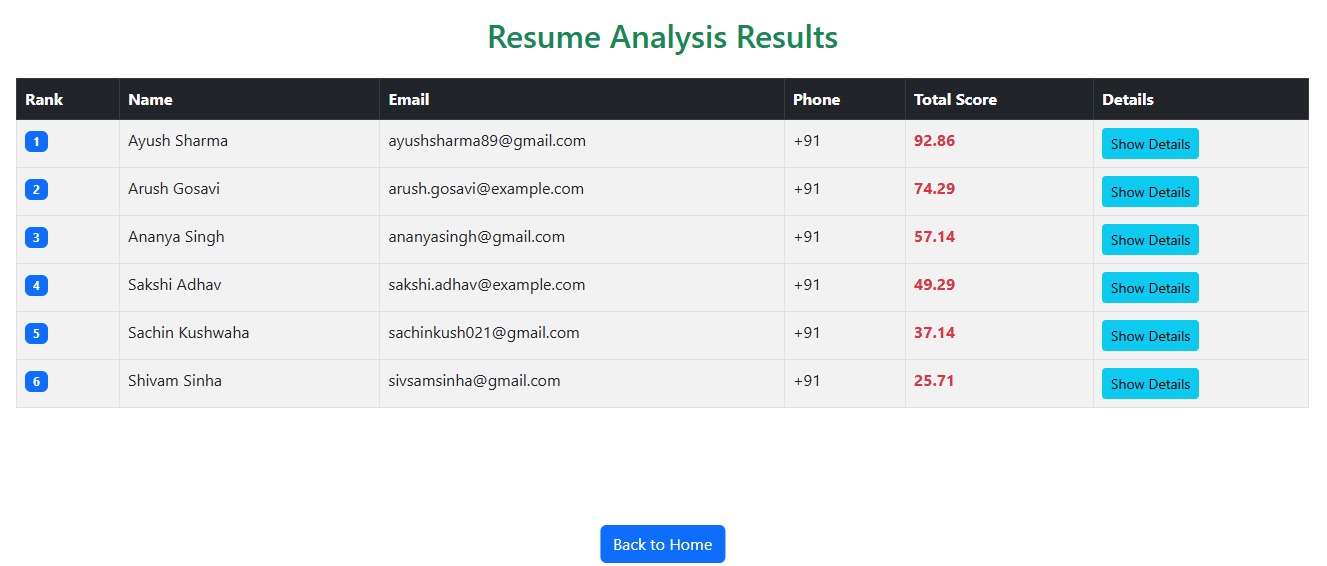


Image 6: Resume Analysis and Result



Flow Chart

