**Resume and JD Parsing and matching system**

**Design Document for Resume Matching and Scoring System**

**Overview**

This document details the design of a system that matches and scores resumes against a given job description (JD). The system processes and extracts information from resumes and JDs, and then utilizes a matching engine and scoring algorithm to evaluate the compatibility of applicants for a specific role.

1. **Architecture Diagram**

A diagram of a computer program

Description automatically generated

**B. Component Design**

**a. Job Description Parser**

**Purpose**: Extracts relevant information from job descriptions.

**Process**:

* **Input**: PDF files of job descriptions.
* **Extraction**: Utilizes pdfplumber to extract text.
* **Parsing**:
  + Uses spaCy to process text.
  + Extracts required skills and expected roles using a Matcher object.
  + Leverages a custom list of job titles and skills from external files.
* **Output**: Structured data including skills and roles required for the job.

**b. Resume Parser**

**Purpose**: Analyzes resumes to extract key information.

**Process**:

* **Input**: PDF files of resumes.
* **Extraction**: Employs pdfplumber for text extraction.
* **Parsing**:
  + Processes text with spaCy.
  + Extracts candidate's skills, experience, name, email, and phone number.
  + Calculates total experience by parsing dates and computing the duration.
  + It also uses external files – Universities.csv, skill.txt for matching skills that are mentioned in the Resume.
* **Output**: Detailed profile of each candidate including experience, skills, and personal information.

**c. Matching Engine**

**Purpose**: Matches resume profiles against job description requirements.

**Process**:

* **Input**: Structured data from both job description and resume parsers.
* **Matching Logic**:
  + Compares extracted skills from resumes with required skills from JDs.
  + Considers additional factors like role designation and total experience.
* **Technologies**: Utilizes advanced NLP techniques and spaCy matchers for efficient matching.
* **Output**: A list of resumes matched against each job description based on relevant criteria.

**d. Scoring Algorithm**

**Purpose**: Scores each matched resume based on its relevance to the job description.

**Process**:

* **Input**: Matched resumes from the Matching Engine.
* **Scoring Criteria**:
  + **Skill Overlap**: Measures the overlap between candidate’s skills and JD's required skills.
  + **Experience Relevance**: Assesses if the candidate’s total experience aligns with the job's requirements.
  + **Role Suitability**: Evaluates if the candidate's previous roles align with the expected role.
* **Algorithm**:
  + Uses BERT embeddings (transformers library) to encode skills and roles for semantic comparison.
  + Employs cosine similarity to measure the closeness of resume and JD embeddings.
  + Scores normalized between 0 to 100 for interpretability.
* **Output**: A score for each resume indicating its suitability for the job.

**Deployment Documentation for Containerizing and Deploying the Resume-JD Matcher as a Microservice**

This document guides you through the process of deploying the Resume-JD Matcher Python application as a Dockerized microservice. The application involves parsing resumes and job descriptions (JD), matching skills, and calculating experience.

**Prerequisites**

* Docker installed on your system.
* Basic knowledge of Docker and containerization.
* Python 3.x environment.
* Access to the source code of the Resume-JD Matcher.

**Step 1: Preparing the Python Application**

1. **Structure Your Application**: Ensure your Python application is structured correctly. The project directory should contain all necessary Python scripts, data files, and a requirements.txt file listing all dependencies.
2. **requirements.txt**: This file should include all Python dependencies needed for your application, including spacy, pandas, pdfplumber, transformers, torch, etc.

**Step 2: Create a Dockerfile**

Create a Dockerfile at the root of your project directory. This file is used to build a Docker image for your application.

Dockerfile

# Use an official Python runtime as a parent image

FROM python:3.8-slim

# Set the working directory in the container

WORKDIR /usr/src/app

# Copy the current directory contents into the container at /usr/src/app

COPY . .

# Install any needed packages specified in requirements.txt

RUN pip install --no-cache-dir -r requirements.txt

# Run app.py when the container launches

CMD ["python", "./main\_script.py"]

Replace ./main\_script.py with the main entry script of your application.

**Step 3: Building Your Docker Image**

In your project directory, build the Docker image using the following command:

bash

docker build -t resume-jd-matcher .

**Step 4: Running Your Docker Container**

Run the Docker container using:

bash

docker run -p 4000:80 resume-jd-matcher

Adjust the port mappings as necessary for your application.

**Step 5: Deploying as a Microservice**

For deployment, Kubernetes or a similar orchestration tool can be used.

**5.1 Kubernetes Deployment**

1. **Create a Deployment Configuration**: Write a deployment.yaml that defines your deployment, service, and any other necessary resources.
2. **Apply the Deployment**: Use kubectl apply -f deployment.yaml to deploy your containerized application onto a Kubernetes cluster.
3. **Manage and Scale**: Utilize Kubernetes commands to manage your deployment and scale it according to demand.

**Additional Considerations**

* **Volumes**: If your application requires persistent storage, define volumes in your Docker and Kubernetes configurations.
* **Environment Variables**: Manage sensitive data or configuration settings using environment variables.
* **Logging and Monitoring**: Set up logging and monitoring to track the performance and health of your application.
* **CI/CD Pipelines**: For production deployments, integrate your application with CI/CD pipelines for automated testing and deployment.