Technical Guide

Project Name: Resume Analyzer

Authors: Darragh McGonigle (18316121), Stephen McAleese (18921756)

Supervisor: Michael Scriney

0. Table of contents

1. Introduction

1.1. Overview

Resume Analyzer is an online web application designed to help software engineers understand the software engineering job market better. Users can upload and analyze their

resumes and find valuable information about the current software engineering job market that can inform the user on their current level of readiness in the job market and how they could improve their skills to become more employable. The application extracts insightful information about the job market by analyzing large numbers of job posts and summarizing the statistics as intuitive visualizations. This information is also compared to information in the user's resume to inform the user on how well their skills match current job market requirements. Users can upload their resumes to identify the current level of demand for their skills in the job market and find jobs that match their skills. The resume analysis dashboard also shows a list of job posts that match the user's skills, recommended skills that are similar to the user's existing skills, and an overall resume score.

In addition to analyzing their resumes, users can use the reports page to view information about the current job market such as the most in-demand skills, a distribution of the number of years of experience required in the job market, and the locations with the most job opportunities. Finally, a user can view a graph diagram that shows common career roles in the software industry, how qualified the user is for each role, and job posts related to each role.

1.2. Motivation

Finding a new job as a software engineer can be a challenging process. Despite the best efforts of job candidates, the probability of each job application resulting in a job offer is often low. For software engineers, the job application process usually involves several steps, and submitting a resume is often the first step in the job application process.

There is a lot of information online about how to write persuasive and effective resumes. However, in addition to articulate and high-quality writing, most software engineering job postings require the applicant to have specific technical skills. Though the principles of good resume writing do not vary substantially between job posts, the technical skills required by each job post often do.

By reading through job postings on job sites such as Indeed, a software engineering job candidate can get some idea of the kind of skills that they should learn or add to their resumes. However, the number and variety of skills that may be required are large and it is difficult to get an idea of which skills have the highest demand from reading job postings.

We developed ResumeAnalyzer to address this problem. ResumeAnalyzer aggregates and analyzes thousands of job posts to give software engineering job candidates a better idea of which technical skills have the highest demand and which skills might be most valuable to learn as well as many other useful features.

2. System design

2.1. System architecture

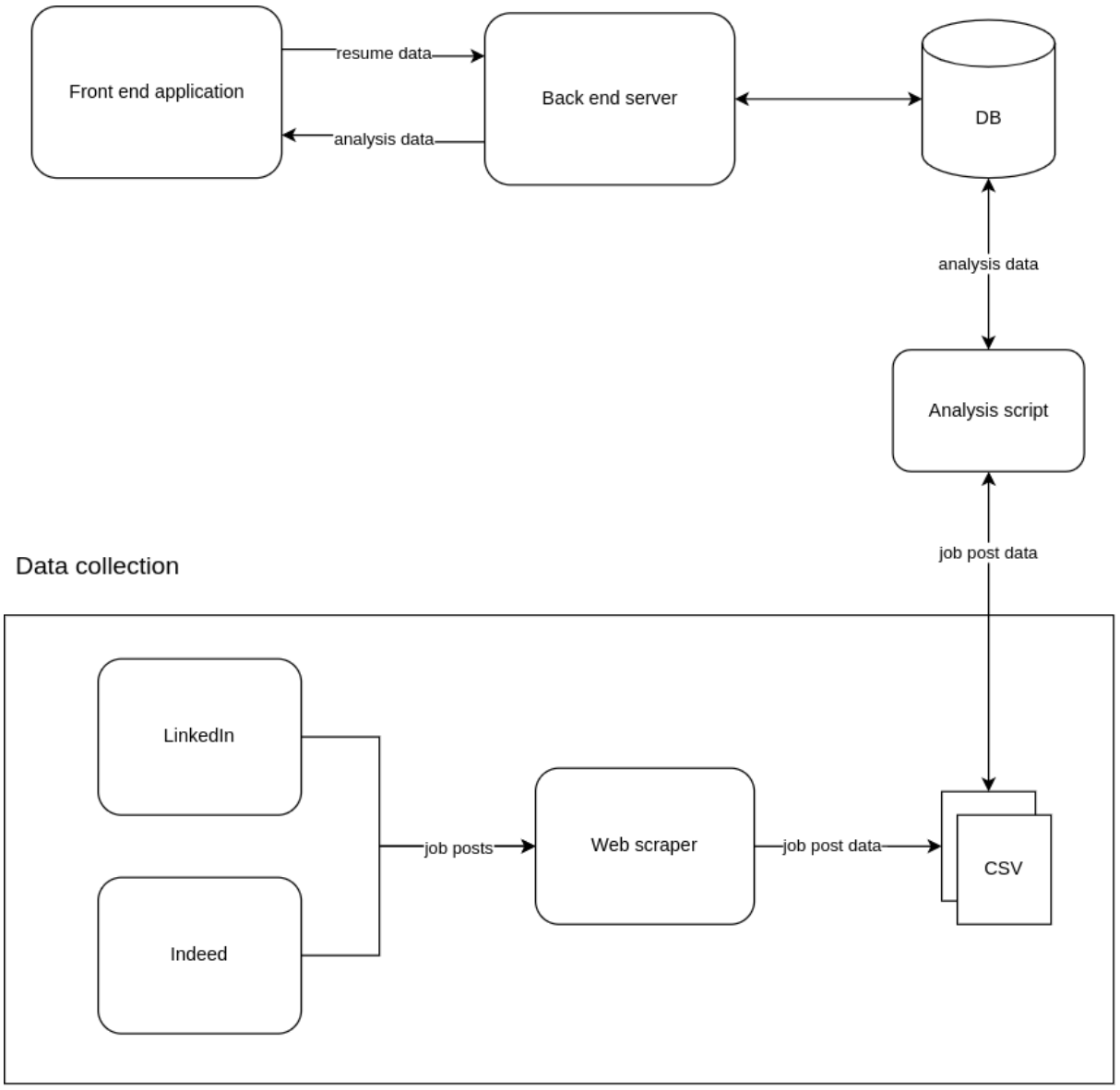

Figure 1: ResumeAnalyzer system architecture

ResumeAnalyzer is an online web application composed of three primary components: the frontend web app, the backend server, and a web scraper. There are two web scraping scripts: one for scraping jobs from LinkedIn and a second script for scraping jobs from Indeed. When the scraping scripts are run, they gather information about job posts from the job sites and save the job post information in CSV files.

As there are a large number of job posts and analyzing them takes a long time, it is inefficient and too slow to analyze the job posts every time a user makes a request to analyze a resume. To improve performance, the job posts are analyzed in advance and the

analysis data is stored in a SQL database so that it can be retrieved quickly for users. A Python script named ‘populate-database.py’ reads the CSV files, analyzes each job post, and extracts information such as a list of skill requirements in the job post, the years of experience required by the job post, and the type of job role that is being advertised.

Each job post and its associated analysis information is then inserted into a SQL database. When a user uploads a resume, the job post analysis information is read from the database and used to efficiently analyze the resume. The results of the resume analysis are then sent back to the app so that the user can view them.

2.2. Technology choices

The frontend web application is implemented using the ReactJS library and the react-bootstrap library is used for some of the UI elements such as tables and buttons. We chose these libraries because they are well documented and popular throughout the industry. Also, React's component-based UI management and state management functionality were suited for implementing the project’s complex and dynamic UI.

The backend service powers the frontend application by responding to its requests. It is written in Python and the FastAPI library. Some popular backend Python libraries include Django, Flask, and FastAPI. We chose FastAPI as it is simple and modern while still being well-documented. The database we chose was PostgreSQL because it is a popular, mature, and open-source technology. We also used the SQLAlchemy ORM to interact with the database instead of using raw SQL queries to increase productivity and reduce the chance of bugs. However, raw SQL was sometimes needed for more complex queries.

For data analysis, we used Python packages such as NLTK, scikit-learn, NumPy, and gensim in addition to Python standard library data structures and algorithms such as dictionaries, arrays, and regular expressions.

2.3. Deployment

ResumeAnalyzer is a web app running live at http://www.resumeanalyzer.xyz. The frontend application, backend application, and database operate independently and communicate with each other over the internet. The frontend React application is deployed as a Heroku application. The backend server that powers the frontend application is another Heroku application and the two applications interact via HTTP network calls. The PostgreSQL database is deployed as a Heroku add-on to the backend application. It is running on a server and communicates with the backend application over the internet.

3. Implementation details

3.1. Scraping scripts

The Indeed and LinkedIn scraping scripts are written in Python and use the BeautifulSoup and Selenium libraries. The scripts operate by entering search queries into the Indeed or

LinkedIn job websites, making requests for job posts, and scraping information from each job post. Since each job uses a standardized HTML format, the scraping scripts are able to identify and extract information from the HTML by identifying HTML elements and classes and extracting text from these elements. Job posts that contain invalid or no information are skipped. The scripts output the data extracted from each job post into a CSV file. Since there are two scraping scripts and one CSV file for each script, the scraping process results in two CSV files which each contain job post data.

3.1. Data analysis script

A Python script named ‘populate-database.py’ takes the two CSV files as input and inserts the job post data into the SQL database. The script also analyzes the job posts and inserts additional columns to store the analysis information.

The CSV files have the following headings: id, company, job title, location, and description. In addition to these columns, the data analysis script also adds the following additional columns: requirements, experience, and role.

3.1.1 Extracting requirements

The requirements column is a list of skill requirements for each job post and the list is created from the job post description. To do this, an initial list of about 100 skills from ‘skills.csv’ is loaded into the script. Each skill keyword also has related keywords so that the keyword identification process is more effective. For example, the Javascript framework ‘Express’ is a skill, and the related keywords ‘ExpressJS’ and ‘Express.js’ will also result in a match for that skill.

To identify the skills from the job post description, stopwords such as ‘the’ and ‘a’ are removed from the text to increase efficiency. Then, for each word in the description, the script checks if the word is in the set of skills. If it is, a new skill requirement is added to the list of requirements associated with the job post.

3.1.2 Skill counts

Every time a skill requirement keyword is found in a job post, the overall count for that skill is also incremented and the counts for each skill are then stored in the database for efficient retrieval later.

3.1.3 Role

For each job post, the analysis script also computes the role name that matches the description best using the k-means machine learning algorithm. For example, a description might match the “Software Engineer” role or the “Software Engineering Manager” role.

3.2. Database

We are using the PostgreSQL database which can be run online as an add-on to a Heroku application. We are also using the SQLAlchemy ORM (object-relational mapping) to interact

with the database. The ORM makes it possible to interact with the database using Python code instead of raw SQL queries. Though SQL may be needed for advanced queries.

There are three database tables in the database ‘job_post’, ‘rule’, and ‘skill’. The job_post table stores information about job posts, the skill table stores the skill counts found by analyzing the job posts and the rule table is used to find skills related to the user’s existing skills.

The Python script ‘populate-database.py’ creates these tables and populates them with rows using information from the CSV files and analysis information produced by the data analysis process.

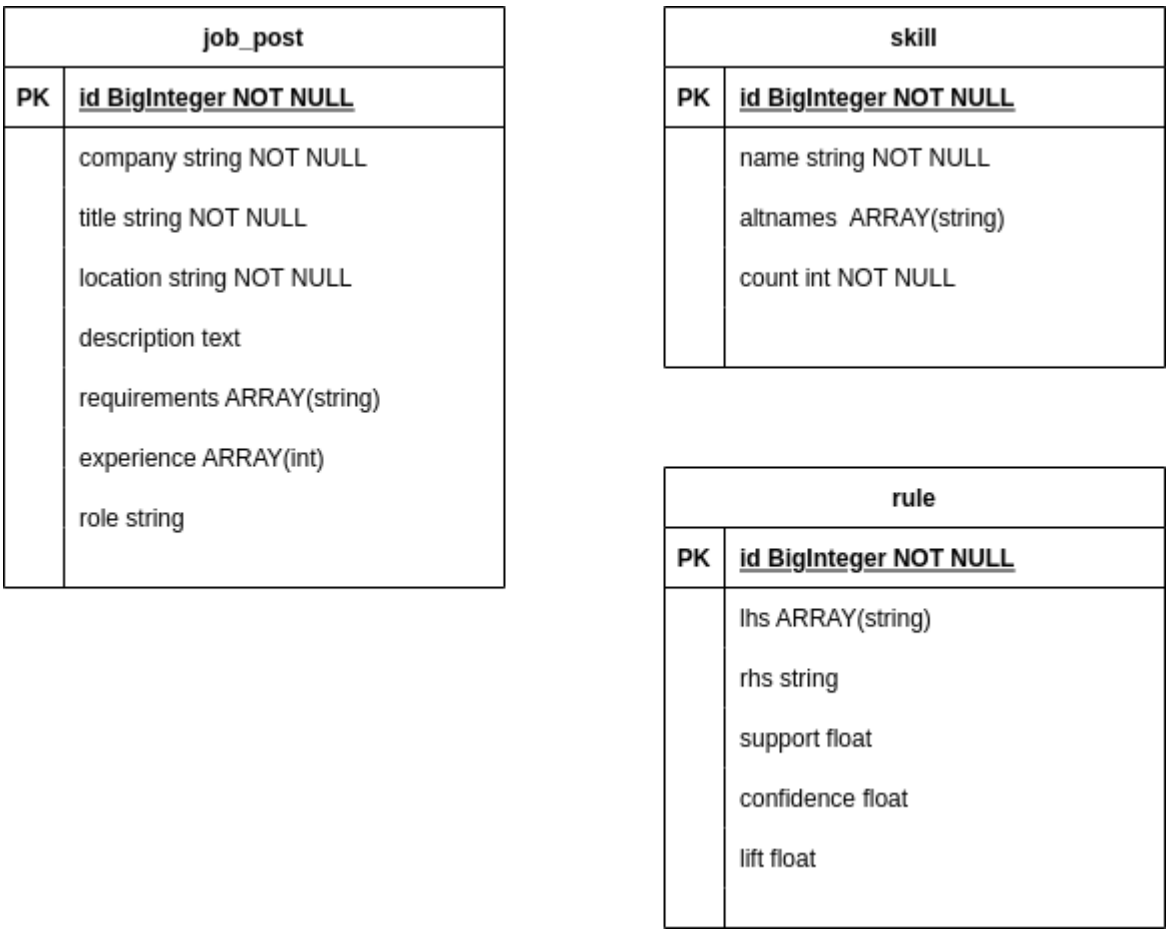

Figure 2: database entity-relationship diagram

3.2. Frontend application

The frontend application was written using the ReactJS Javascript library. Additional libraries or packages are installed using npm (node package manager) and these packages are stored in the package.json file. Some libraries which are commonly used in the frontend application include ‘styled-components’ which makes it possible to style React components more directly using CSS and ‘react-bootstrap’ which provides a suite of pre-styled

components such as tables, buttons, and cards. The chart-js package is used to create charts in the application.

3.3. Backend service

The backend service is written in Python and uses the FastAPI library for handling network requests. The frontend application makes requests for information to the backend service. The service is organized as a REST API and uses HTTP methods such as GET and POST to send and receive information. REST endpoints are defined in the ‘server.py’ file. The application retrieves information from the database and the database connection is defined in ‘database.py’. Some common database functions are defined in ‘crud.py’. The ‘pydantic’ Python package is used to define the attributes and types for objects that are going to be inserted into the database. These types are defined in ‘schemas.py’ and ensure that only valid objects are added to the database.

4. Problems and solutions

4.1. Extracting text from resumes

One of the earliest challenges we faced when creating this project was the problem of extracting text from PDF resumes. Initially, we did not have the ability to extract text from PDFs but after some experimentation, we found that the pdfplumber Python library was highly effective at extracting text from resumes.

4.2. Identifying skill keywords from job posts

A core feature of our web app is the ability to identify skill keywords from job post descriptions and count the keywords to find the relative frequencies of skills in the job posts. Our initial solution was to train a neural network on tagged StackOverflow posts so that it could then identify skill keywords from the job posts. However, we found that this approach had two major problems: suboptimal precision and performance.

Precision is a metric that can be defined as the percentage of a system's output that is correct. We found that our initial AI model tended to inappropriately identify some words from job posts or resumes as skills resulting in a lower precision. Although the AI model was fast enough for identifying skill keywords from resumes, we found that it was too slow for identifying skill keywords from hundreds of job posts.

To increase precision and performance, we instead created a list of around 100 skill keywords. To count the skills in a job post description, our current implementation first removes stop words from the description before looping through each word in the description. If a word is in the skill keywords set, a counter variable associated with that skill keyword is incremented. This new implementation is much faster than the previous implementation. It also has higher precision because a description word will only be identified as a skill keyword if it has been explicitly defined as a skill keyword in the list of

skills. Another advantage of this new and simpler approach is that, since it is deterministic, its output can be tested more easily.

4.3. Database performance problems

As we added more features to our web app and analyzed the job posts in greater detail the number of database requests required to analyze each job post increased significantly resulting in performance issues. As the number of database calls increased, we found that it took an increasing amount of time to analyze the job posts and populate the database, especially when populating the online production database which has more network latency than the local database.

To address this performance deficiency, we refactored the job post analysis and database code so that it could do the same work with much fewer database requests. The solution was to store more information in memory to reduce the number of database requests needed. For example, we found that it is much faster to update skill counts in a local dictionary instead of updating the database every time a new skill keyword is found. Once the in-memory skill counter dictionary has been fully updated, only a few database calls are needed to update the database with all the information in the local dictionary.

5. Testing

4.1. Unit testing

Unit testing determines if individual functions in a program behave as expected.

We wrote unit tests for the Python backend using the unittest Python library. The tests were written to test important functions in the backend code responsible for tasks such as extracting skill keywords from resumes, extracting skill keywords from job posts, and extracting years of experience requirements from job posts. Each unit test tests a single function and in each test, an input and an expected output are defined. The expected and actual output are then compared. If the expected and actual output are the same, the test passes and fails if there is a difference.

4.2. Integration testing

The purpose of integration testing is to determine whether several software modules interact with each other as expected.

4.2.1. Backend integration tests (API tests)

We created backend integration tests using the Postman API testing library. Our Postman tests make requests to the server and check whether the server responds as expected. These API tests are integration tests because, in order for the tests to pass, the server, several functions, and the URL need to work correctly together to produce the correct response.

4.2.1. Frontend integration tests

For the frontend, we created integration tests using Reacting Testing Library which test how multiple user interface components interact with each other. The purpose of these tests is to ensure that the app UI changes as expected when a user takes action. For example, when the app is loaded, it should show the home screen and when the user uploads a resume, the UI is expected to show certain panels.

4.3. Continuous integration tests

Continuous integration is the practice of regular merging new code changes into a central repository and running automated tests on the new code to ensure that it doesn't break existing functionality.

To implement CI tests, we used the GitLab CI runner. The runner can be programmed by defining what the runner should do in the .gitlab-ci.yml file. In our project, the CI runner builds the app before running the API tests, integration tests, and unit tests.

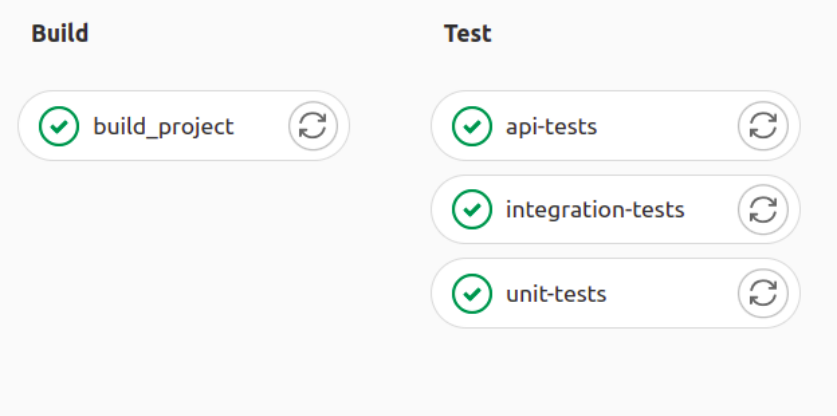

Figure 3: CI runner setup

4.4. User testing

We carried out user testing to test the usability and usefulness of our app. We created a user testing survey with Google Forms and sent it to several other users. The form asks participants to test the online web app and give feedback on how useful they think the app is and how the app could be improved.

4.5. Load testing

We used Apache benchmark to test the performance of our system. The following command sends requests to our server, measures how long each request takes, and saves the results in a CSV file:

ab -n 1000 -e benchmark.csv http://www.resumeanalyzer.xyz/home

Using this approach we found that it takes an average of 68ms to load the home page of our web app even when a large number of requests are sent to the server which suggests that the server can scale to handle many user requests.

6. Future work

During the development of this project, we had the opportunity to implement most of the features we planned to develop. However, there are some features and extensions we did not have time to develop. If we had more time, we might have extended the project with the following features:

- Scrape and analyze job posts from all other the world instead of only job posts in Ireland.
- Gather more statistics and useful information from the job posts such as salary information.
- Measure changes in the popularity of skills between locations and changes in popularity over time.
- Run the server on faster hardware so that the web app can be used by many users all over the world.