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 find jobs and 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.

Lastly users can view a visual diagram of typical software career paths and the various skills needed to progress their careers. Here a user can also view job listings by role type to help users with interests in getting into specific roles in the field find their ideal jobs.

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. Research

2.1 Domain Research

Before any work could be carried out on the project we had to carry out research into the software jobs domain. To do this we started by devising some research questions we had to answer. These research questions were “How do software professionals find jobs?”, “What skills are widely used in the industry?”, “Are there any existing solutions?” and “How can we improve the experience?”. We primarily used Google’s search engine to carry out this stage of research as we knew it would give us access to a wide array of different results and perspectives.

What we found was software developers in Ireland primarily use online job boards such as “Indeed”, “Glassdoor” and “LinkedIn”. They also found jobs through targeted recruitment on social platforms like “LinkedIn”. This research would prove crucial as it gave us a starting point for data gathering for our project (See Section 2.2.1**).** We also found that a few of these platforms had some form of feature or tools for recommending jobs based on users' data. However none of the existing solutions we could find focused on extracting information directly from the user’s resume instead relying on manual data entry to be carried out by the user. On top of this these solutions would just recommend a user's jobs without providing any additional insights into how a user can develop their skill set to become more employable. With this we were able to devise the ways in which we could set our product apart and provide new and novel information to our users.

2.2 Component Level Research

2.2.1 Data Sources and Web Scraping

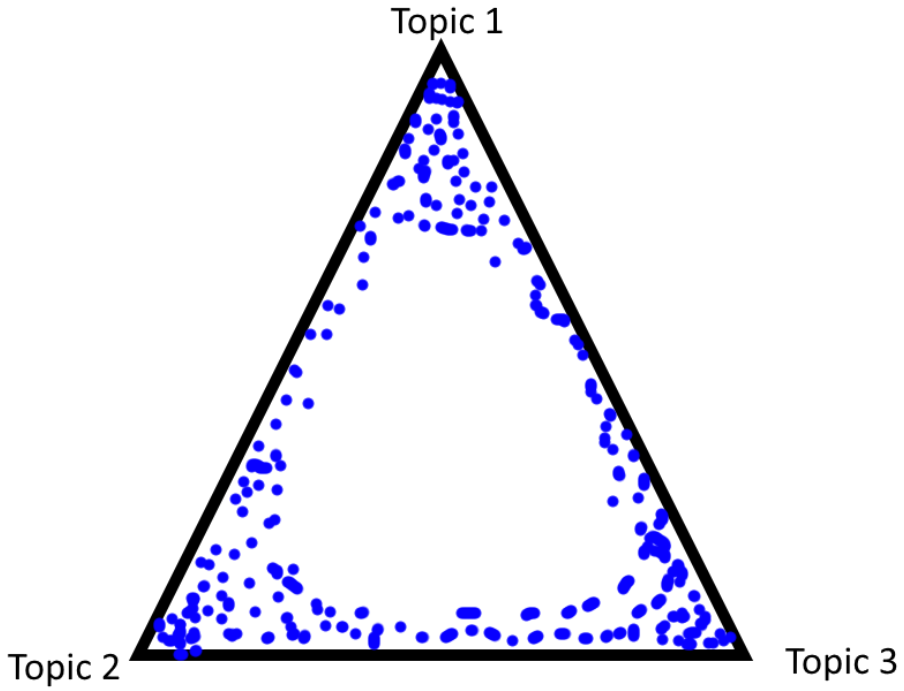
Without our project being focused heavily around data mining we first needed to find sources or data that we could utilize. As mentioned in section 2.1 “Domain Research” we had found a lot of job boards that were widely being used by Irish software professionals and decided to research potential APIs which these platforms might make available. We found that “Glassdoor”, “Indeed”, “Stack Overflow” and “LinkedIn” all had APIs for accessing job posts on their job boards, However none were public and we had to request access to each which none of the 4 companies were willing to grant.

With this revelation that we would not be able to get API access for any data sources we instead opted to look into the process of using web scraping to gather data from these sources. We wrote some test scripts and found that it was possible to scrape both “LinkedIn” and “Indeed” but there were some potential roadblocks such as rate limitation that we would have to account for in our implementation. For more detail on the exact process we undertook in exploring and creating the web scrapers see section 4.4**.**

2.2.2 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is an unsupervised NLP technique for document classification. It works by taking a collection of documents and generating a corpus from all the words in these documents. It then uses this corpus to create a specified amount of pseudo documents, these documents will each contain various amounts of the words from the corpus. A document can then be processed by the model and assigned a vector based on its distance (difference) to each of the pseudo documents.

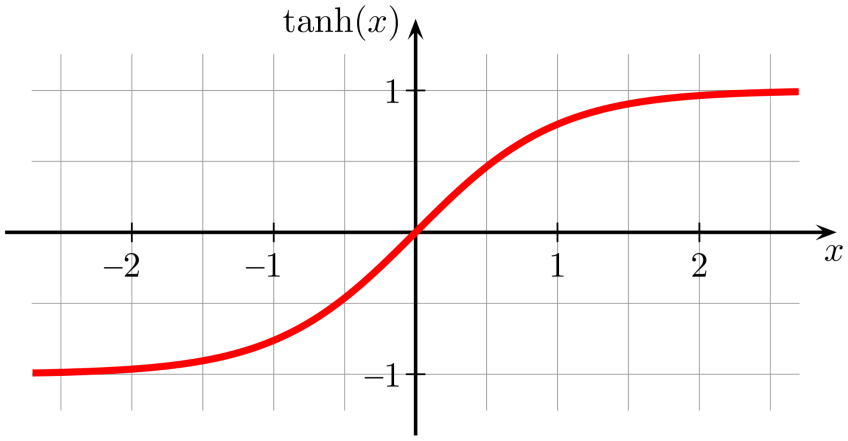
The whole process can be conceptualized as a triangle (or n-dimensional triangle) where each corner of the triangle is a pseudo document. Given a collection of documents to process the resulting vectors will all lie within the triangle and form a Dirichlet distribution (see below). This is advantageous as the nature of a Dirichlet distribution means that resulting vectors are biased towards the edges and thus you get few documents which are equally similar to all pseudo documents.



2.2.3 Score Assignment and Heuristics

We decided that we need to devise a Heuristic to score a user's resume with this however would require some research into how something as subjective as resume quality could be quantified. The approach we took for the final implementation can be seen in section 3.3.3 but in this section we are going to look at two crucial bits of research we undertook for the creation of the heuristic.

Firstly when calculating the skill score we needed a way to normalize the number to be between 0 and 1. To do this we looked at various activation functions and decided to go with tanh or the hyperbolic tangent function (see below). With this we were able to achieve the desired results.



Secondly we wanted to score the resume by length, this required research to be done into the optimal resume length using Google we found the most common consensus to be that 600-700 words was optimal, anything greater was too wordy and anything less lacked detail. We used this to form the basis for our normal distribution curve.

3. System design

3.1. System architecture

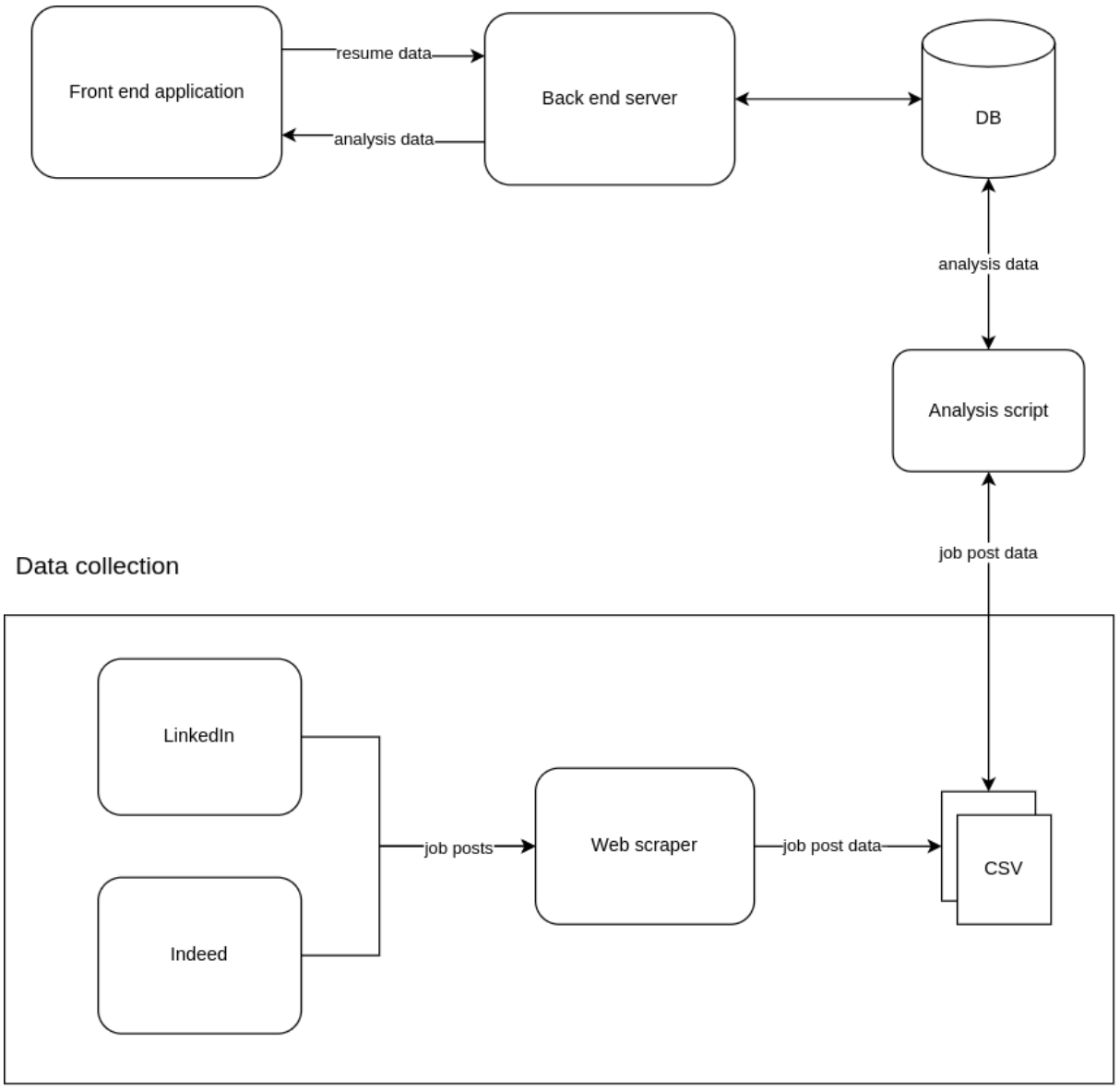


Figure 1: ResumeAnalyzer system architecture

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.

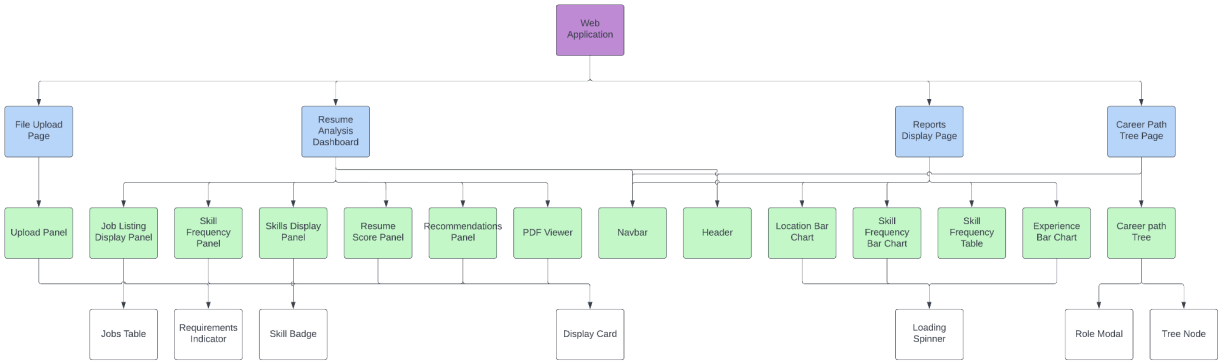
As there are a large number of job posts and analyzing them takes a long time, it is inefficient and too slow to analyze the job posts every time a user makes a request to analyze a resume. To improve performance, the job posts are analyzed in advance and the

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.

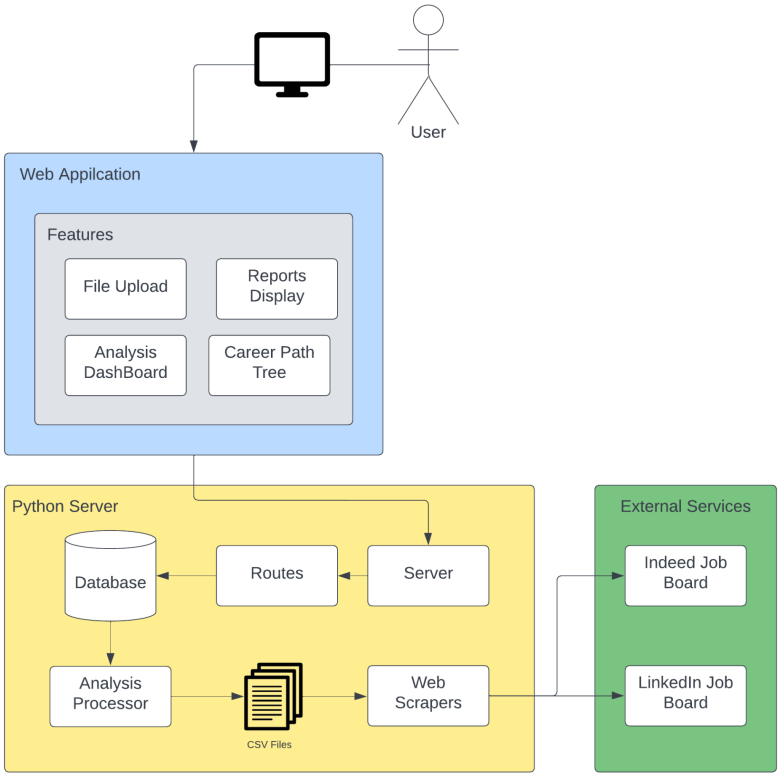
Each job post and its associated analysis information is then inserted into a SQL database. When a user uploads a resume, the job post analysis information is read from the database and used to efficiently analyze the resume. The results of the resume analysis are then sent back to the app so that the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

3.2 High Level Design Diagrams

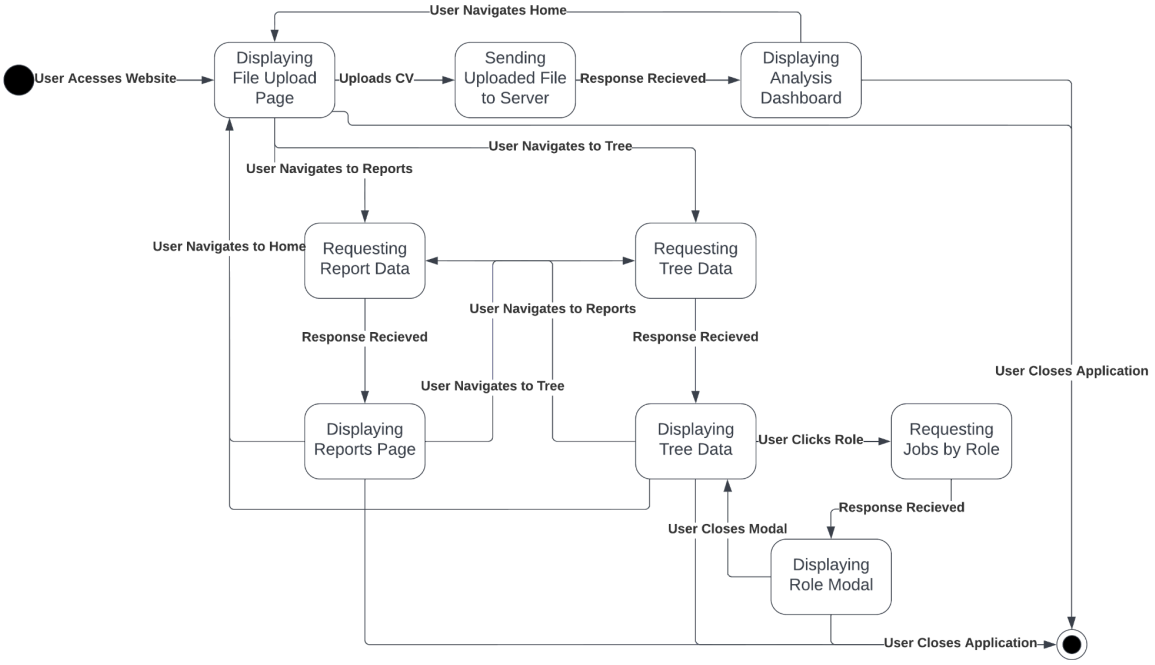
3.2.1 Frontend Component Relationship Diagram



3.2.2 Architectural Overview Diagram



3.2.3 State Diagram



3.3. 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.

3.4. 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.

4. Implementation details

4.1. 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.

4.2. 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.

4.2.1 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.

4.2.2 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.

4.2.3 Role

For each job post, the analysis script also computes the role name that matches the description best. It does this by using a trained LDA (Latent Dirichlet Allocation) model that vectorises the job descriptions. Then K-means clustering is used to group similar job

description vectors, the titles and descriptions of each document in the clusters are then processed for keywords in order to assign a role to the cluster.

4.2.4 Experience

The system is able to determine the required years of experience from a job post by using pattern matching to search for a wide array of potential string patterns. For example a description may contain the pattern ‘X years of experience’. If a match is found the experience value will be extracted and stored in the database alongside the post.

4.2.5 Rules

After the initial pass which processes the job posts and extracts the requirements the system applies Apriori’s Algorithm on the set of requirements with a minimum support value of 0.03. Apriori’s Algorithm then gives us a list of all association rules which can be extracted from the list of requirements that have a support (Frequency of Occurrence / Length of Dataset) which are greater than 0.03.

4.3. 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.

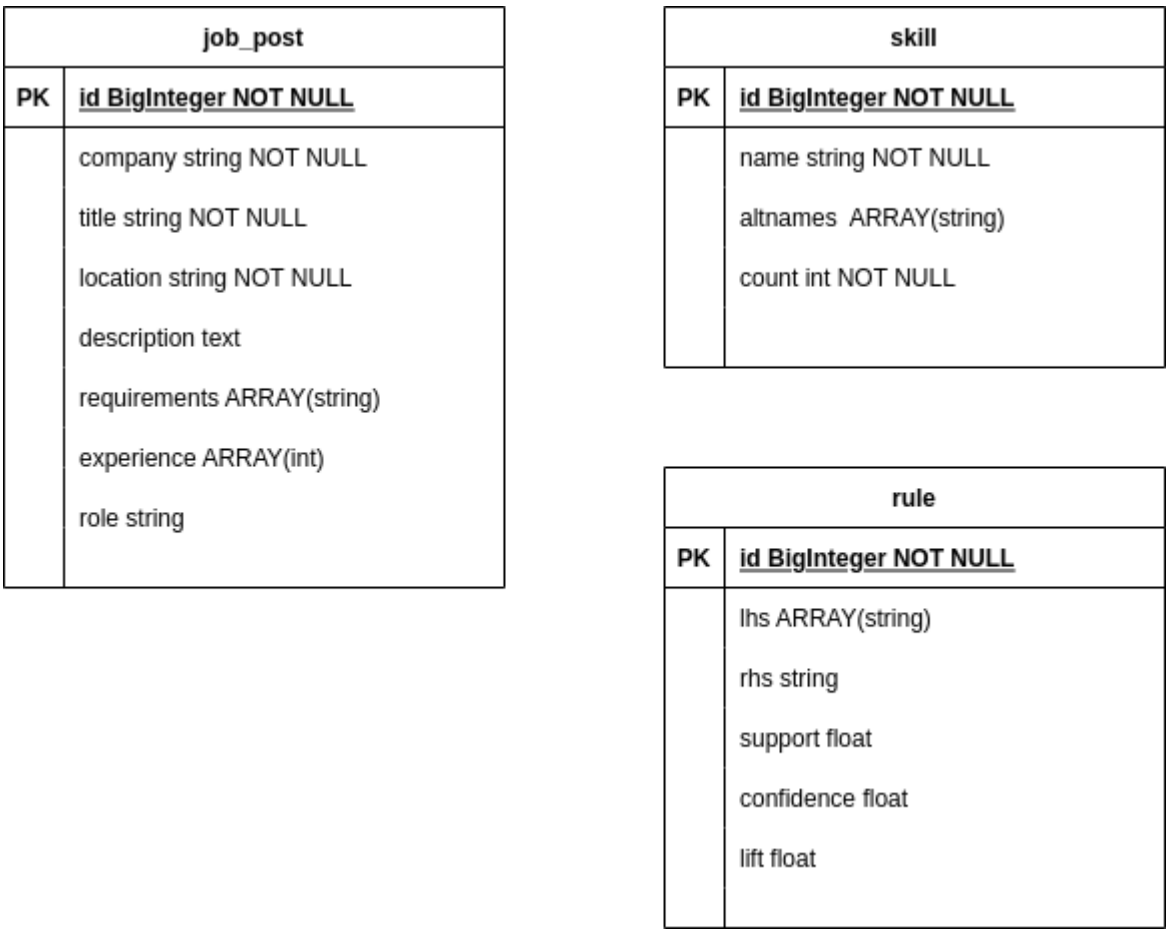


Figure 2: database entity-relationship diagram

4.4. 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.

4.4.1 Navigation

The application’s navigation is handled using a library called ‘react-router-dom’ which enables us to define routes within our application that render different components. This is primarily used in two key places, the first is when a user uploads a file the ‘useNavigate’ hook is used to direct the user to the dashboard display upon receiving a response from the server. The second is with the custom styled ‘react-bootstrap’ navbar which runs along the left edge of the page, on this navbar are various icons which when clicked on by a user will navigate to the corresponding page.

4.4.2 File Upload Page

The file upload page serves as the home page for the web application and is the first thing a user will be greeted with upon entering the site. It is very simplistic in design and contains a prompt for the user to upload a file. The user has two choices on how they wish to upload a file, firstly then can click the highlighted button which will open their computer's file explorer, secondly they can drag and drop a file into the indicated area. The drag and drop functionality was accomplished using ‘react-dropzone’ and the ‘useDropzone’ hook which allowed us to define the page area in which a user could drag a file. The selected file is then sent to the server using a fetch post request.

4.4.3 PDF Display Panel

On the right hand side of the analysis dashboard is a PDF file display which shows the user's uploaded document. This was implemented using the Mozilla PDF worker and the ‘react-pdf’ library. It supports pagination through a ‘react-bootstrap’ pagination component with custom state management.

4.4.3 Resume Skills Panel

This is a simple panel which takes the server's response and looks at the ‘skills’ heading in the response object and maps through the skills using the ‘color’ values to dictate the background colour for each skill name.

4.4.4 Skill Frequency Panel

The skill frequency panel displays the count of the number of occurrences for the top 50 most common skills that have been extracted as requirements from our job posts data. It represents the data as both a doughnut chart from ‘react-chartjs-2’ and as a table.

4.4.5 Matching Jobs Panel

This panel shows all jobs from our database in which the user’s current skill set satisfies all of the job posts requirements. The requirements are visualized using coloured circles where the colors match the colors of the skills shown in the ‘Resume Skills Panel’, the user can also hover over these circles and the name of the skill will be displayed. Each row can also be clicked on which will drop down and show the job posts description; this is done using the ‘react-bootstrap’ accordion component.

4.4.6 Resume Score Panel

The resume score panel consists of three charts; one doughnut chart from ’react-chartjs-2’ which shows the final overall score, and two progress bars from ‘react-bootstrap’ which show how well the resume scored on the two metrics of skills and length.

4.4.7 Skills Recommendations Panel

Shown in the skills recommendations panel are the most applicable rules from the database (this is calculated using the rules lift. Left of the arrow shows the users existing skills which serve as the bases for the recommendation, this side can contain multiple rules and they will be separated by a plus symbol. Right of the arrow is the recommended skill that relates to the left hand side.

4.4.8 Reports Page

The reports page contains graphs and tables which give insight into the data that is stored in the database without relying on data from the user's resume. The four displays are ‘Years of Experience’, ‘Job Post Skill Frequencies’, ‘Job Post Location’ and ‘Job Post Skill Frequency Distribution’. The top of the page contains a search feature which allows the user to filter the page to show only charts that match the search query.

4.4.9 Career Path Tree Page

The reports page contains graphs and tables which give insight into the data that is stored in the database without relying on data from the user's resume. The four displays are ‘Years of Experience’, ‘Job Post Skill Frequencies’, ‘Job Post Location’ and ‘Job Post Skill Frequency Distribution’. The top of the page contains a search feature which allows the user to filter the page to show only charts that match the search query.

4.5. 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.

4.5.1 Resume Skill Extraction

The system uses ‘pdf-plumber’ to extract the raw text of the uploaded resume. This text is then tokenized using ‘spacy’ and then those tokens are filtered to remove stop words using the ‘nltk’ stop words package. Lastly all the remaining tokens can be checked against the skills database to see which are skills and thus should be extracted.

4.5.2 Skill Colour Assignment

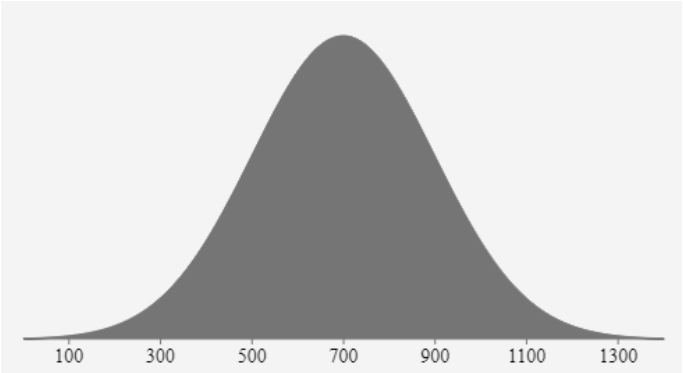
All extracted skills are assigned a colour for frontend display purposes. This is done in the Hue-Saturation-Value (HSV) colour space, a hue value is assigned to each skill by dividing 360 by the amount of skills to have all the skills be evenly spaced around the colour wheel. Lastly the same saturation and value values are added to each of these hue values which ensure the colours are bright and that text will be legible.

4.5.3 Resume Score Calculation

Resume score is a heuristic which takes into account two characteristics of a user's resume. Firstly the skills from the resume are each taken and a skill score value is incremented by the count of the skill in the database divided by the average count of skills in the database. This resulting skill score is then fed into the tanh activation function and multiplied by 100 to give a skill score out of 100.

$$\text{Skill Score} = \tanh(\sum \text{skill\_count} / \text{average\_skill\_count}) * 100$$

The second resume characteristic is resume length which is calculated using a normal distribution with a mean of 700 and a standard deviation of 200 (This comes from research into optimal resume word counts). The word count of the user's resume is then plugged into this distribution and the resulting value is multiplied by 100 to get a value between 0 and 100 called the length_score.



The final score is calculated by taking a weighted average of 2:1 in favor of skill score.

5. Problems and solutions

5.1. 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.

5.2. 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 based ‘Named Entity Recognition’ (NER) model 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.

5.3. 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.

5.4 Data Gathering

Initially we had hoped to be able to utilize APIs from job boards such as ‘Indeed’, ‘Stack Overflow, ‘Glassdoor’ and ‘LinkedIn’ but all of these API providers declined to give us access to their APIs. Without a source of data the whole project would not have been possible as all of the features relied heavily on data.

To solve this problem we started by each writing a web scraping script for one of the aforementioned job boards. We chose ‘Indeed’ and ‘LinkedIn’ as they had the largest quantity of job listings. We used ‘Selenium’ and ‘Beautifulsoup 4’ in order to scrape the sites. These scraping scripts revealed some potential issues with rate limiting and when writing the finalized version of the scrapers we had to add sleep calls to ensure our requests were not blocked.

Although API access would have been a much cleaner and simpler solution that would have been able to provide more data faster, this web scraping solution was able to get the required data that we needed to be able to push on with the project.

5.5 Role Classification for Job Posts

In order to create the career path tree we needed to divide a way of classifying job posts into defined roles. Initially we had considered doing some form of supervised machine learning but dismissed this idea as ~2000 job posts would be very time consuming to read through and manually classify in order to create a training set.

In the end our supervisor introduced us to the concept of Latent Dirichlet Allocation (LDA) which is an unsupervised NLP method for vectorising documents based on their similarity to ‘n’ number of pseudo documents. We trained up an LDA model with 18 pseudo topics, with this model we then vectorised all of the job descriptions.

Now that we had the documents in a vectorised form we had to work out how to classify that vector as a single role. First we tried the most simple approach which was to assign the documents based on which pseudo topic it was closest to (index of lowest value in vector). This did not end up producing the best results because a lot of documents were on the boundaries between topics.

The next approach we tried was supervised regression, we manually classified the 1000 of the 2000 documents and created a regression model with the document vector as the input and the classification as the output. This method performed very poorly due to a very small training data set.

The final classification method we tried was ‘K-Means clustering’ where ‘K’ number of virtual points are computed (centroids) and vectors could then be classified based on which centroid they were closest to. This method ended up being the best of the 3 and had the added bonus that it gave flexibility with the number of different classifications without having to retrain the LDA model and adding additional time to the vectorisation step.

Now that we had a way of grouping documents we then needed to assign groups to a role. We did this using regex and matching to look for keywords in the titles and descriptions which would indicate what role they belong to.

This solution was not perfect and we believe that the regression model would be a better solution had we had more time and more data to create an adequate training set. That being said, the solution we do have in place is designed in a way so that if given a larger dataset it can be easily retrained to provide better results without the need for supervised classification.

5.6 Career Path Tree Qualification

For the career path tree we wanted to create a way to visually represent which roles a user is and isn't qualified for and how close a user is to being qualified for a role. We decided to do this by applying a gradient to the tree nodes background which when full would indicate that the user is qualified. This qualification is based on 2 factors, the user needs to have half of the top identified skills and be qualified for at least one of the roles directly above the role.

The initial implementation of this would generate the tree then find each of the node elements and check the immediate parent nodes and then re-render the tree with the styling. This however did not work as the library we are using to display the tree ‘react-flow’ makes heavy use of the ‘useMemo’ hook for performance improvement and does not re-render on the update of node styles. To solve this we devised an algorithm that for each node would check all parent nodes recursively allowing us to eliminate the need to re-render the tree.

6. Testing

6.1. 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.

6.2. Integration testing

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

6.2.1. 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.

6.2.1. 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.

We also created a Newman automated runner for these tests so the whole suit of API tests could be run automatically from the command line.

6.3. Continuous integration tests

Continuous integration is the practice of regularly 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.

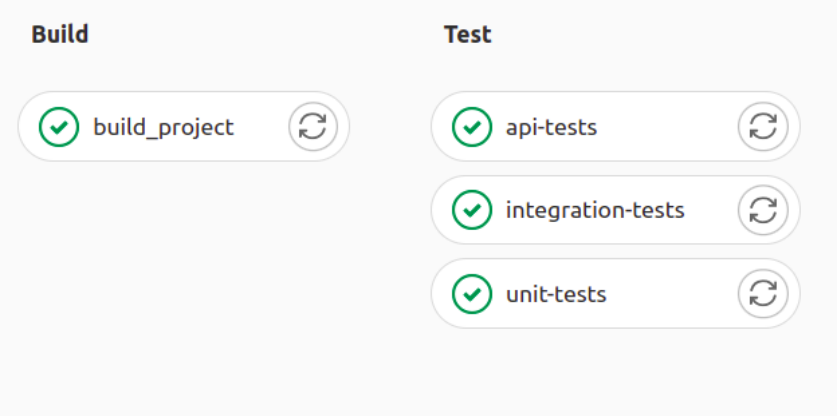


Figure 3: CI runner setup

6.4. 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.

6.5. 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.

7. 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.