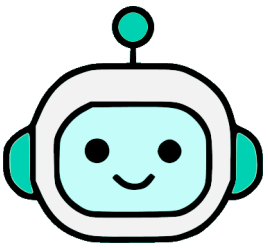

Technical Guide

Project Name: ResumeAnalyzer

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

Supervisor: Michael Scriney

Date Completed: 24/04/2022

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 to inform them about 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.

Lastly, users can view a visual diagram of typical software career paths and the various skills needed for each path. Here a user can view how qualified they are for certain career roles and also view job listings by role type to help users who are interested in getting into specific roles in the field.

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 only reading job postings.

We created 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 we started the project we had to carry out research on the software jobs domain. To do this we started by devising some research questions 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 to the problem we are seeking to solve?” and “How can we improve the experience of finding new jobs?”. 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 that software developers in Ireland primarily use online job boards such as Indeed [1], Glassdoor [3], and LinkedIn [2] to find jobs. They also find jobs through targeted recruitment on social platforms such as 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 have tools for recommending jobs based on users' data [4]. However none of the existing solutions we could find focused on extracting information directly from users’ resumes and instead relied on manual data entry to be carried out by the user. On top of this, these solutions tended to recommend jobs to users without providing any additional insights into how users could develop their skills to become more employable. Knowing these limitations, we were able to devise ways to set our product apart and provide new and novel information to our users.

2.1.1 Footnotes

1. Indeed: https://ie.indeed.com/
2. LinkedIn: https://ie.linkedin.com/
3. Glassdoor: https://www.glassdoor.ie/index.htm
4. LinkedIn Recommendations: Job Recommendations – Overview | LinkedIn Help

2.2 Component-Level Research

2.2.1 Data Sources and Web Scraping

As our project’s features heavily rely on data, we first needed to find data sources for it. As mentioned in the section “2.1 Domain Research” above, we found job boards that were widely being used by Irish software professionals and decided to research potential APIs that 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 of them were public and we did not succeed in requesting access to any of the APIs.

With the 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 though there were some potential roadblocks such as rate-limiting that we would have to account for in our implementation. To write these web scrapers we used both

BeautifulSoup as it gave us easy HTML parsing and Selenium for the LinkedIn scraper as it allowed us to mimic user interaction which was necessary to get access to job posts through LinkedIn. For more detail on the exact process we undertook for creating the web scrapers see section “4.1 Scraping Scripts“.

2.2.2 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is an unsupervised NLP technique for document classification [1]. 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 word counts from the corpus. A document can then be processed by the model and assigned a vector based on its distance (difference) from each of the pseudo documents.

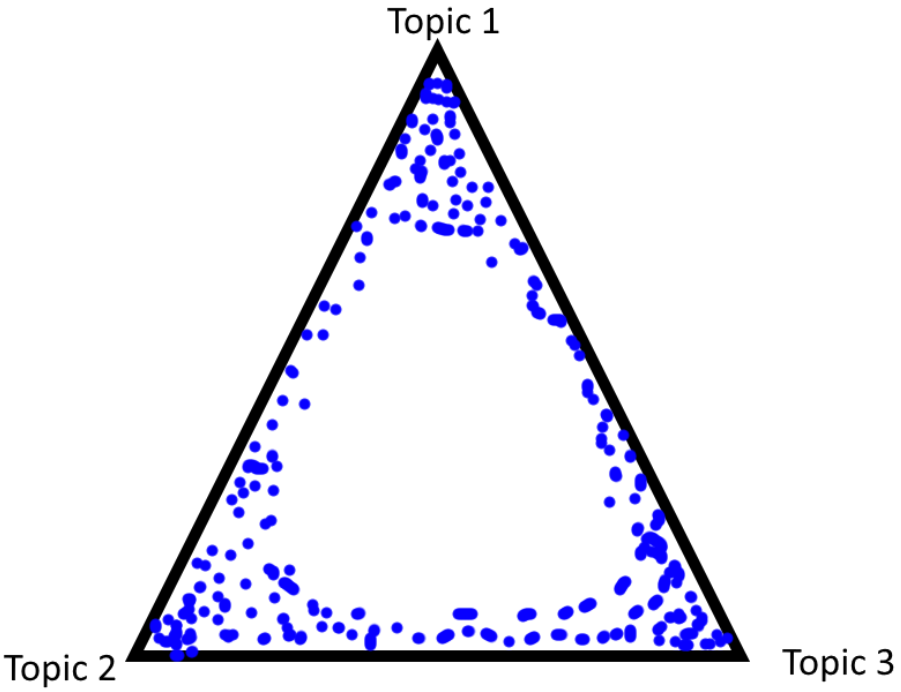

Figure 1: LDA

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 there are few documents that are equally similar to all pseudo documents.

2.2.3 Score Assignment and Heuristics

We decided that we needed to devise a heuristic to score users’ resumes. Before creating one, we first needed to do some research into how something as subjective as resume quality could be quantified. The approach we took for the final implementation is explained in detail in section 3.3.3. In this section, we will describe the research and algorithms underpinning the heuristic.

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

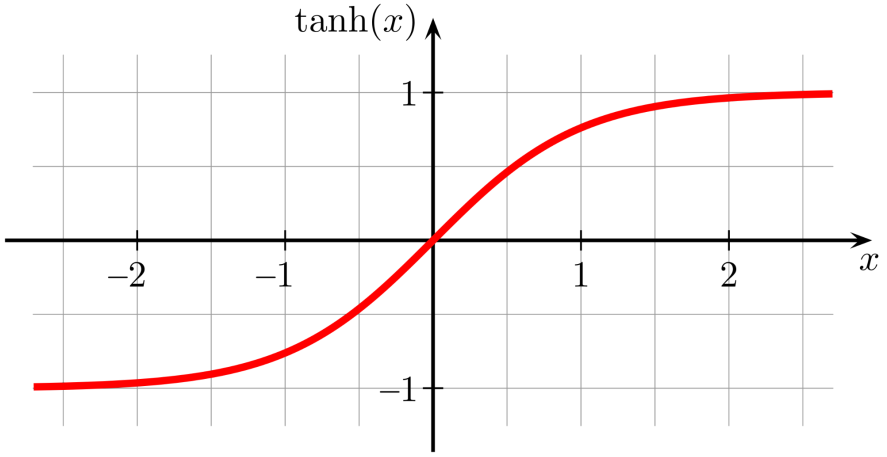

Figure 2: tanh function

Secondly, we wanted to score the resume by length which required research to be done into the optimal resume length. Using Google we found the most common consensus to be that approximately 400-800 words was optimal [2,3], anything longer was too wordy and anything shorter lacked detail. We used this research to create a normal distribution curve. For more information, see section “4.5.3 Resume Score Calculation”.

2.2.4 Footnotes

1. LDA Research Paper: Blei, D.M., Ng, A.Y. and Jordan, M.I., 2003. Latent Dirichlet Allocation. Journal of Machine Learning Research, 3 (Jan), pp.993-1022.
2. Livecareer Optimal Resume Word Count
3. AvidCareerist Optimal Resume Word Count

3. System design

3.1. System architecture

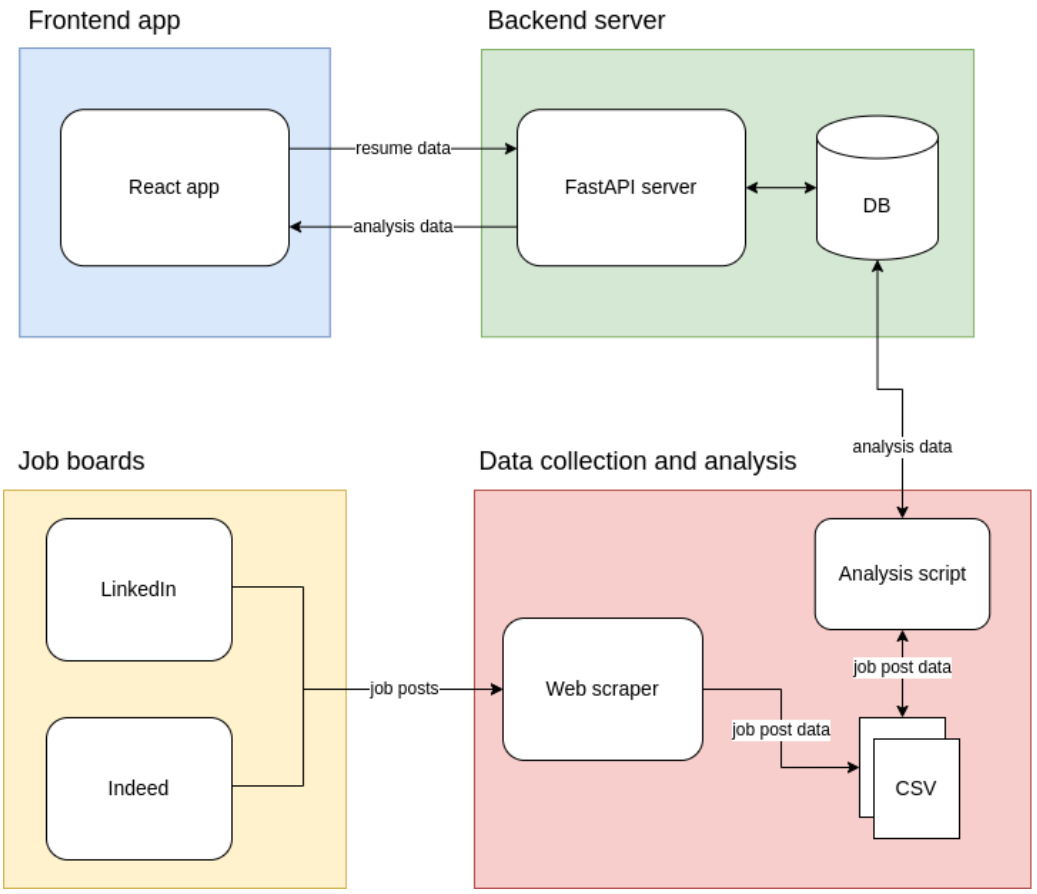

Figure 3: ResumeAnalyzer system architecture

ResumeAnalyzer is an online web application composed of three primary components: a frontend web app, a 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 can take several minutes, it is inefficient and too slow to analyze all 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

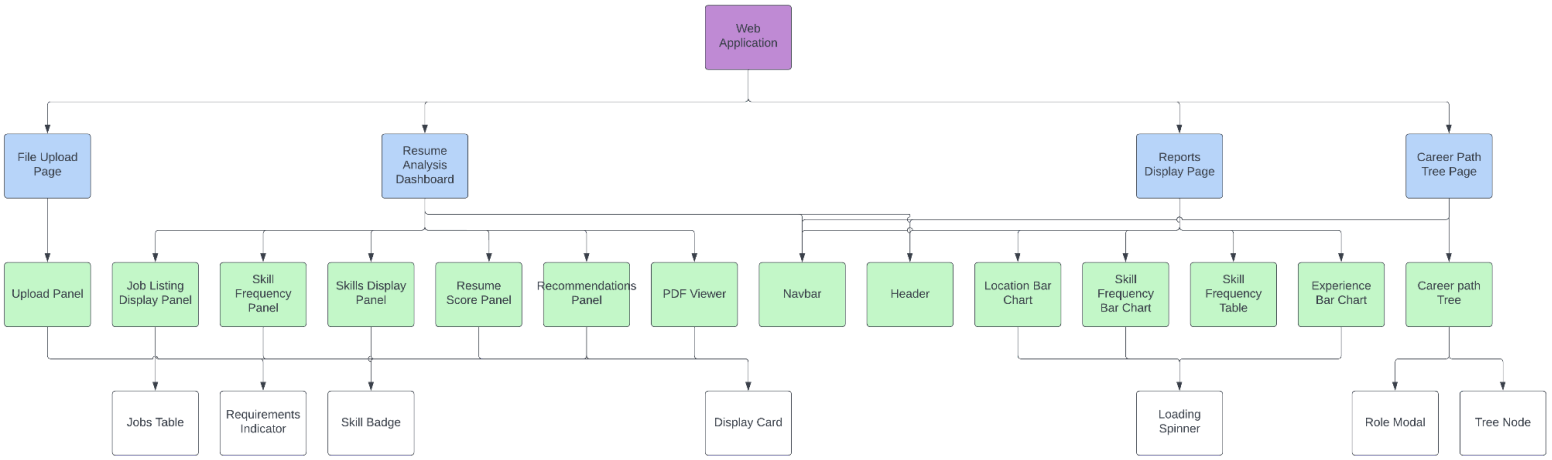

Figure 4: Frontend Component Relationship Diagram

The above diagram shows the component hierarchy for the frontend web application. The very top level is “App” which is composed of 4 pages: “File Upload Page”, “Resume Analysis Page”, “Reports Display Page” and “Career Path Tree Page”. Each page is made up of multiple React components and some components such as “Navbar” are used in multiple pages. Finally, some sub-components used by some of the components are shown in white.

3.2.2 Architectural Overview Diagram

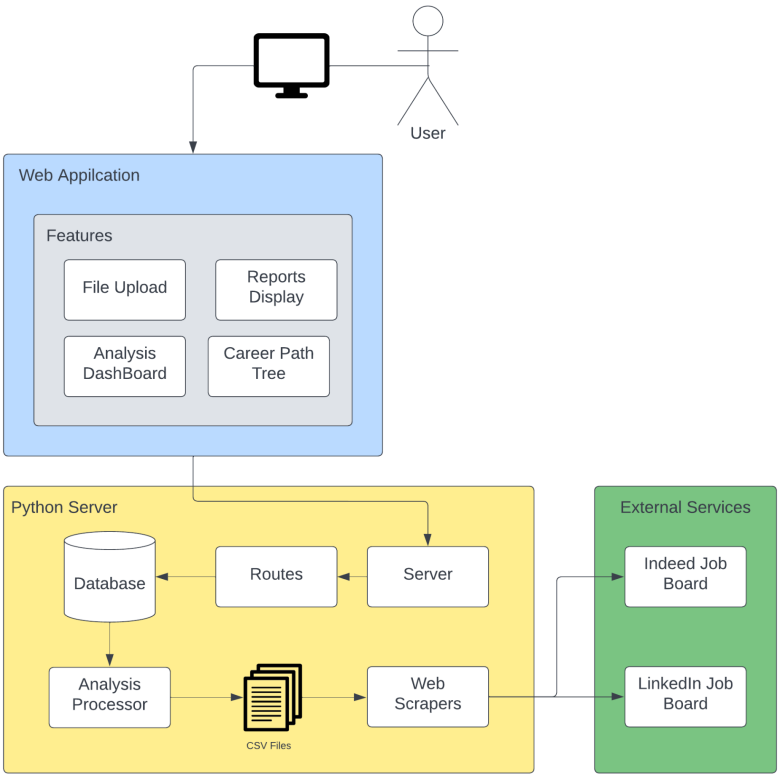

Figure 5: Architecture Overview Diagram

The Architecture Overview Diagram shows the high-level architecture of the application from the user and the web application (and its features) to the backend server and finally the data source.

3.2.3 State Diagram

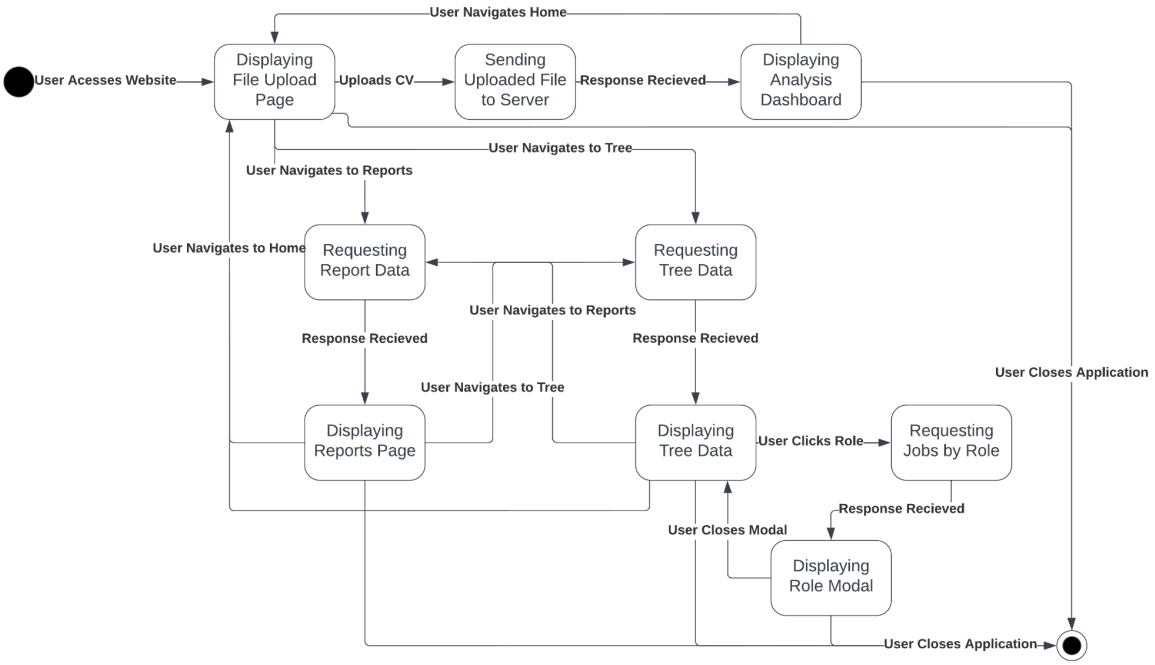

Figure 6: State Diagram

The state diagram shows the state of the application at any given time based on the user's interactions with the application. It starts when the user accesses the website and shows all possible states and transitions that could occur before the user closes the application.

3.3. Technology choices

The frontend web application is implemented using the ReactJS [1] library and the react-bootstrap [2] 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 software industry. Also, React's component-based UI management and state management functionality were well-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 [3] and the FastAPI [4] 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 [6] because it is a popular, mature, and open-source technology. We also used the SQLAlchemy ORM [5] 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 [7], scikit-learn [8], NumPy [9], and gensim [10] 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 https://www.resumeanalyzer.xyz. The frontend React application is deployed on AWS Amplify and the backend FastAPI application is deployed on AWS Elastic Beanstalk. Both applications have custom domains which are more readable than their original AWS domains. The backend makes calls to a PostgreSQL database. The database is hosted in RDS which is another AWS hosting service.

3.5. Foot Notes

1. React: https://reactjs.org/
2. React-Bootstrap: https://react-bootstrap.github.io/
3. Python: https://www.python.org/
4. FastAPI: https://fastapi.tiangolo.com/
5. SQLAlchemy: https://www.sqlalchemy.org/
6. PostgreSQL: https://www.postgresql.org/
7. NLTK: https://www.nltk.org/
8. Scikit-Learn: https://scikit-learn.org/
9. Numpy: https://numpy.org/
10. Gensim: https://radimrehurek.com/gensim/
11. Heroku: https://www.heroku.com/home

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 the job posts into CSV files.

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 into the database 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 that 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.

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 vectorizes the job description. 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 role names to each cluster.

4.2.4 Experience

The required years of experience from each job post description is identified using regular expression pattern matching to search for potential phrase matches. The years of experience digits are then extracted from these phrases. For example, a description may contain the pattern ‘X years of experience’. If a match is found, the years of experience value will be extracted from the match and stored in the database alongside other information about the job.

4.2.5 Rules

After the initial pass which processes the job posts and extracts the skill requirements, the system applies Apriori’s Algorithm to 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 value (Frequency of Occurrence / Length of Dataset) greater than 0.03.

4.3. Database

We are using the PostgreSQL database which is run online as an AWS RDS database instance. 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 four database tables in the database ‘job_post’, ‘rule’, ‘skill’, and ‘soft_skill’. The job_post table stores information about job posts, the skill table stores the skill counts found by analyzing the job posts, the rule table stores information produced by Apriori’s Algorithm to find related skills, and the soft_skill table stores soft skill counts.

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.

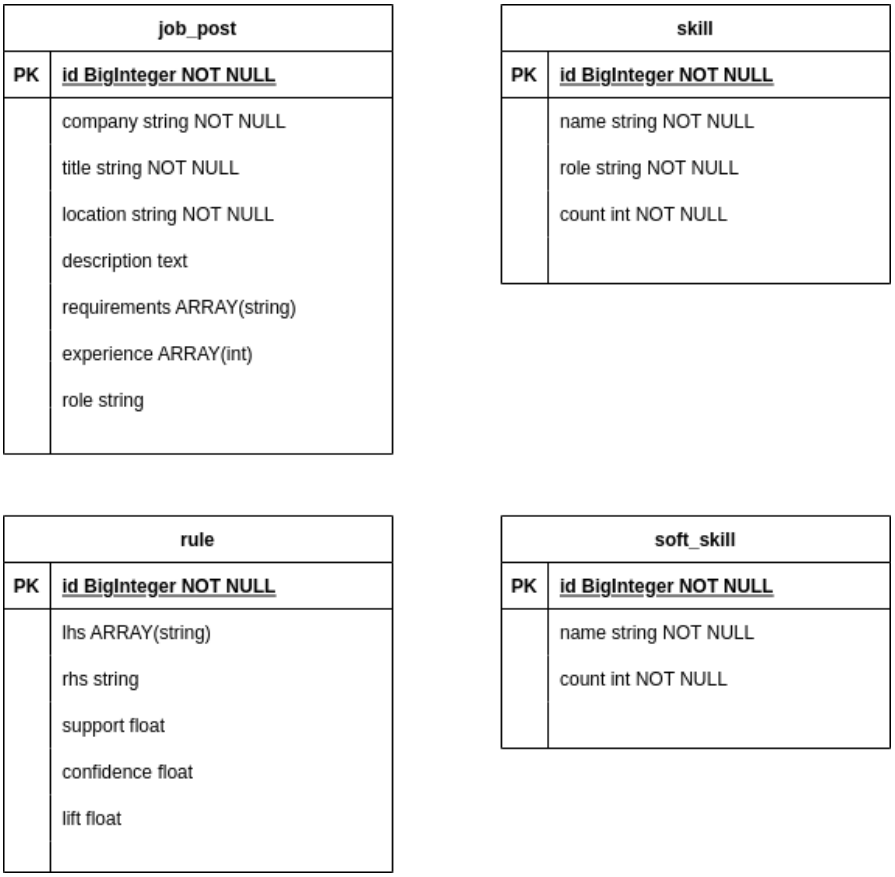

Figure 7: database entity-relationship diagram

4.4. Frontend application

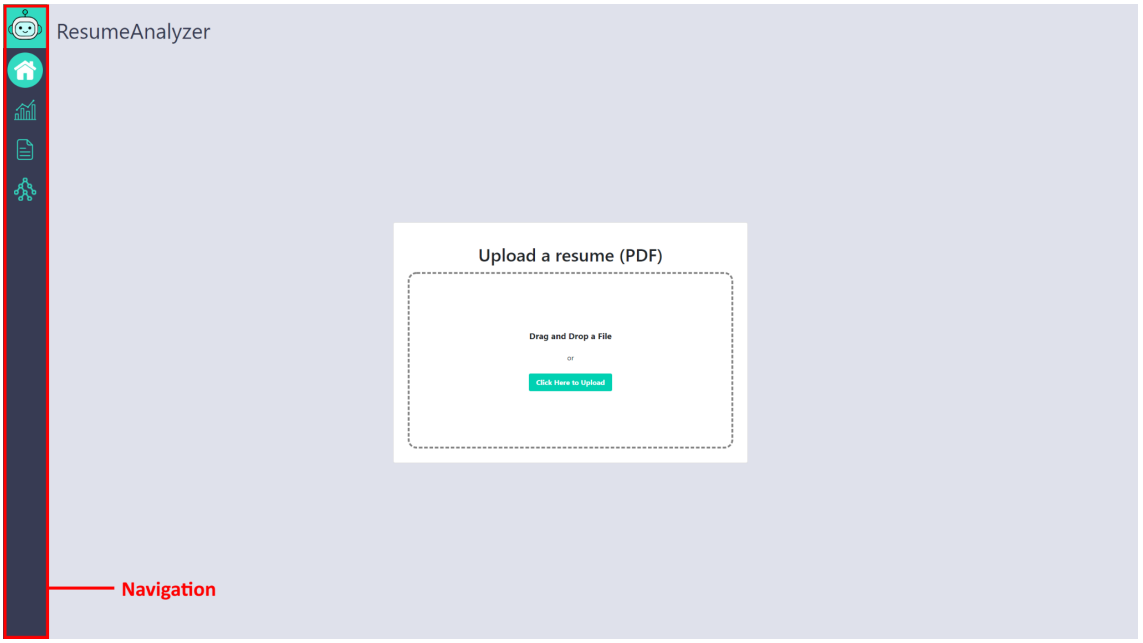

Figure 8: File Upload Page

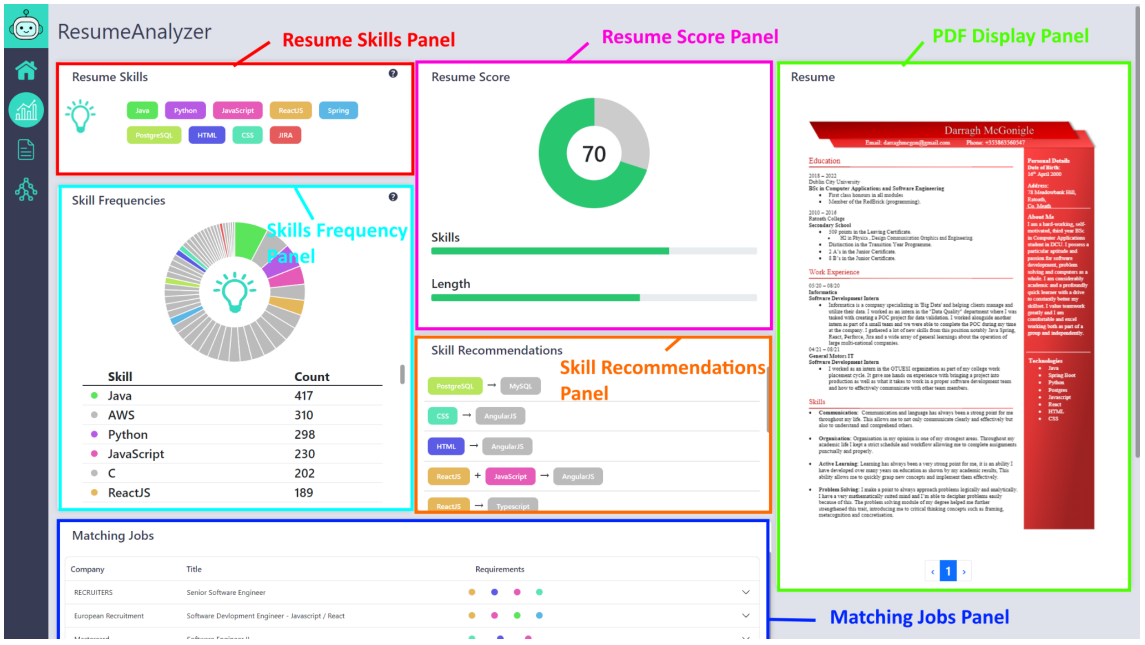

Figure 9: Resume Analysis Page

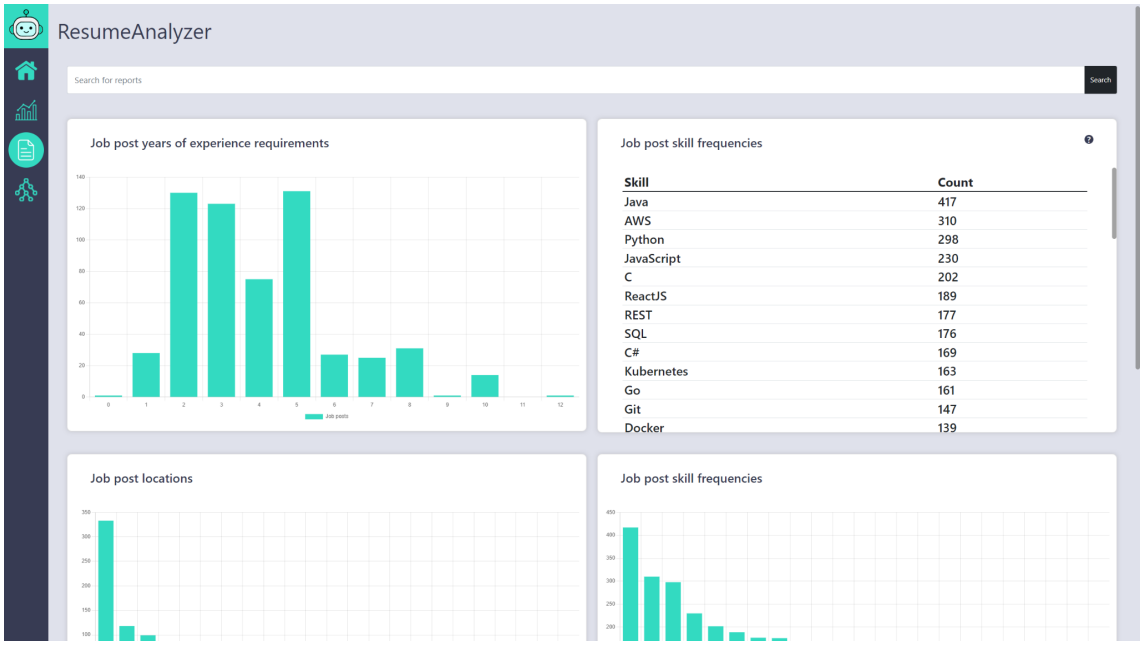

Figure 10: Reports Page

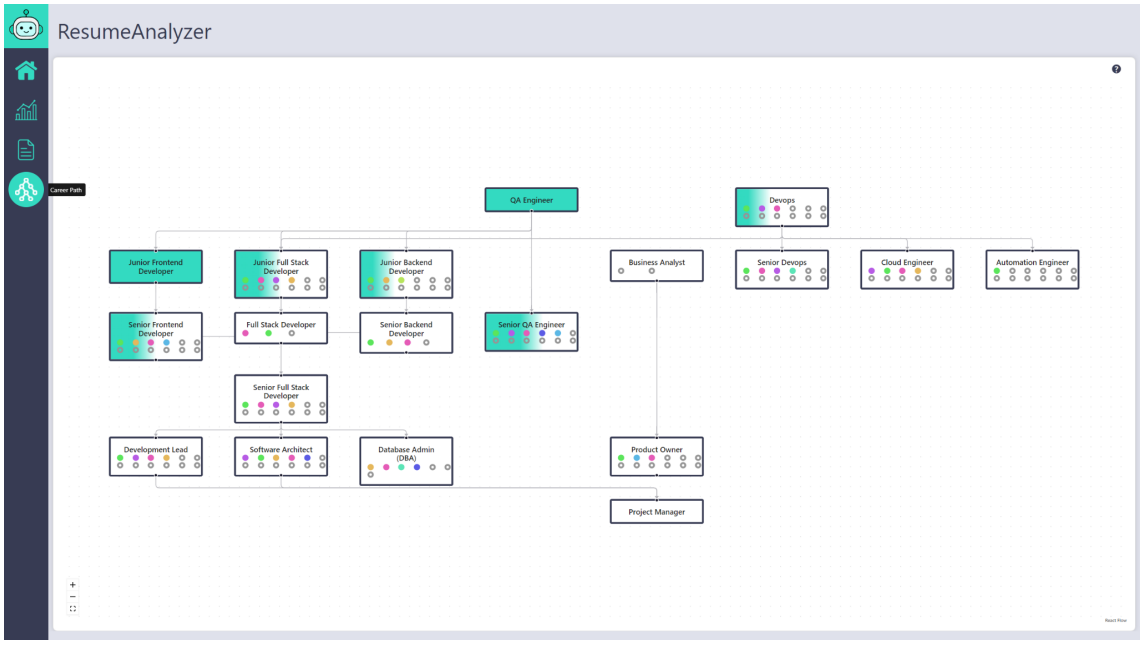

Figure 11: Career Path Tree Page

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 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, first is when the ‘useNavigate’ hook is used to direct the user to the dashboard display after the user has uploaded and submitted a resume. Secondly, with the custom-styled ‘react-bootstrap’ navbar which runs along the left edge of the page. On this navbar are several icons that 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 after loading the web app. 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 they can click the highlighted button which will open their computer's file explorer. Secondly, they can drag and drop a file into the file upload 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 panel that shows the user's uploaded document. This was implemented using a 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 panel simply takes as input the resume skill keywords in the response object from the server and assigns color values to the skill keywords before displaying the colored keywords in a panel.

4.4.4 Skill Frequency Panel

The skill frequency panel displays the counts for the top 50 most common skills extracted from the job posts. It represents the data as both a doughnut chart from ‘react-chartjs-2’ and as a table. Each role has different skill counts and the role can be changed via a dropdown on the panel.

4.4.5 Matching Jobs Panel

This panel shows all jobs from our database where the user’s current skillset satisfies all of the job posts requirements. The requirements are visualized using colored circles where the colors of the circles match the colors of the skills shown in the doughnut chart 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 causing a dropdown to show the job post’s 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. The resume is evaluated according to the skill counts for a particular role and that role can be changed with a dropdown on the panel.

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 rule’s lift value). To the left of the arrow shows the user's existing skills which serve as the bases for the recommendation. This side can contain multiple skills which are separated by a plus symbol. To the right of the arrow is the recommended skill that relates to the left-hand side. Recommended skills are only shown if they do not already occur in the user’s resume.

4.4.8 Reports Page

The reports page contains graphs and tables which provide 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 years of experience requirements’, ‘Job post skill frequencies’, ‘Job post locations’, ‘Job post skill frequencies’, and ‘Job post soft skill frequencies’. The top of the page contains a search feature that allows users to filter the page to show only charts that match the search query.

4.4.9 Career Path Tree Page

The career path tree page displays a graph containing possible career paths within the software engineering field such as “Frontend Engineer” and “QA engineer”. When a user’s resume has been uploaded, the degree to which the user’s skills match each job role’s requirements is shown as a green highlight on each node which increases in width the more the user’s skills match a node’s requirements. Clicking on a node in the graph displays skills and job posts related to that role in a modal.

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 the ‘nltk’ package and then those tokens are filtered to remove stop words. Lastly, all the remaining tokens are checked against a set of all skills. If there is a match, a skill is added to the list of requirements associated with the job post description.

4.5.2 Skill Colour Assignment

All extracted skills are assigned a color for frontend display purposes. This is done in the Hue-Saturation-Value (HSV) color space. A hue value is assigned to each skill by dividing 360 by the number of skills so that all the skill values are evenly spaced resulting in a rainbow of colors. Lastly, the same saturation and value values are used for every hue value which ensures the colors are bright and the text is legible.

4.5.3 Resume Score Calculation

The resume score calculator takes two characteristics of a user’s resume into account. Firstly, for each skill found in the resume, a skill score value is incremented using a weighted value that is equal to the skill count divided by the average skill count for all the skills in the database for that role. The resulting skill score is then fed into the tanh activation function and multiplied by 100 to give a skill score between 0 and 100. The skill scores are calculated for every role because each role has different roles associated with it.

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

The second resume metric is resume length which is calculated using a normal distribution with a mean of 500 and a standard deviation of 250 (this value 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.

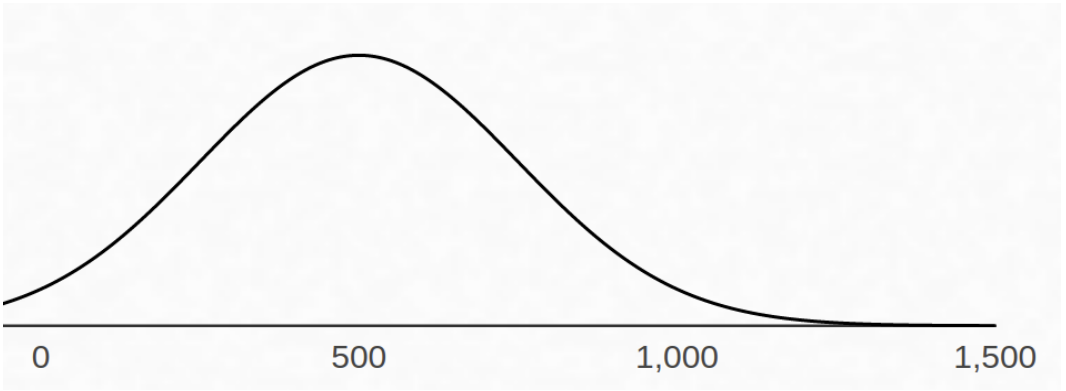

Figure 12: normal distribution for calculating resume length score

The final score is calculated by taking a weighted average of 2:1 in favor of the skill score over the length 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 post collection. 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 post descriptions. 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 words in the system's output that are 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. Also, 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 the hundreds of job posts we needed to process.

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 almost all of the features rely heavily on job data.

To solve this problem we started by writing two scraping scripts: one for Indeed and another for the LinkedIn job board. We choose these sites because they have many job posts. We used the ‘Selenium’ and ‘BeautifulSoup’ Python packages in order to scrape the sites. There were some rate-limiting issues related to using these scripts 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, our web scraping solution was able to give us the data we needed to carry out our project.

5.5 Role Classification for Job Posts

In order to create the career path tree, we needed to find a way of grouping job posts into defined roles. Initially, we had considered doing some form of supervised machine learning but dismissed this idea as annotating thousands of job posts manually would be very time-consuming.

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

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

The next approach we tried was supervised regression: we manually classified 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’ virtual points are computed (centroids) and vectors can then be classified based on which centroid they are

closest to. This method ended up being the best of the three and had the added bonus of being flexible: the number of different classifications could be modified without having to retrain the LDA model or add additional time to the vectorization step.

Now that we had a way of grouping documents we then needed to assign groups to roles. We did this using regex and matching to look for keywords in the titles and descriptions which would indicate which role they belonged 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 such a way 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 users were or weren't qualified for and how close a user was to being qualified for a role. We decided to do this by applying a gradient to the backgrounds of the tree nodes which would indicate how qualified the user was for the role based on how full the gradient was. To be considered qualified for a role, 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 generated the tree, found each of the node elements, checked the immediate parent nodes, and then re-rendered the tree with the styling. This approach did not work however as the library we are using to display the tree named '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 problem 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

See testing-manual.pdf for test run breakdown

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.

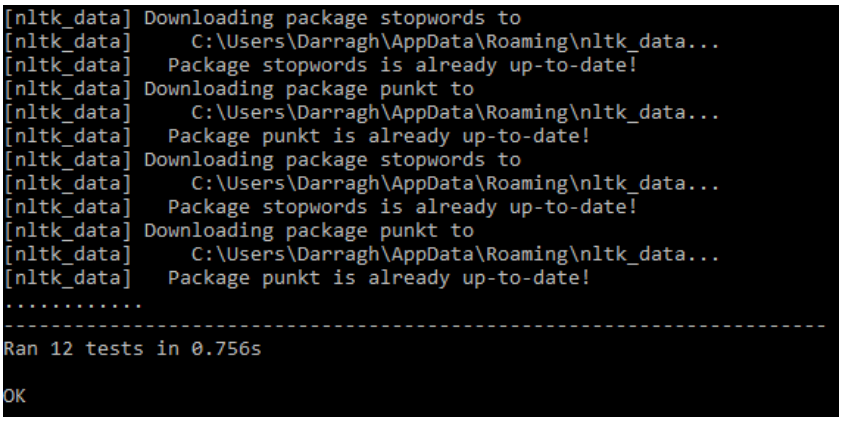

Figure 13: unit test results

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.

	executed	failed
iterations	1	0
requests	7	0
test-scripts	7	0
prerequest-scripts	0	0
assertions	25	0
total run duration: 3.4s		
total data received: 6.31MB (approx)		
average response time: 343ms [min: 31ms, max: 1033ms, s.d.: 393ms]		

Figure 14: integration test results

6.2.2. 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 interacts with it. 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.

```
PASS src/tests/App.test.js (36.211 s)
  Test loading the app
    √ When the app is loaded, the upload page should be shown (87 ms)
  Test resume upload
    √ When a resume is uploaded, the submit button should be shown (91 ms)
    √ When the resume is submitted, the results page should be shown (378 ms)
  Test changing the page using the sidebar
    √ When the reports page icon is clicked, the reports page should be shown (91 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        48.897 s
Ran all test suites.
Done in 51.09s.
```

Figure 15: frontend integration test results

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 GitHub actions to define two scripts: test.yml and deploy.yml. The test script runs the unit tests, integration tests and API tests. When all tests have been run, the deploy script runs automatically to deploy the frontend and backend applications.

6.4. User testing

We carried out user testing to evaluate the usability and perceived value of our application. We created two Google Form surveys [1], one for getting participants' consent as required

under DCU's ethics policy and the second for capturing their feedback about the application. Then we went about receiving ethical approval to carry out the user testing through DCU's ethics panel. With the ethics panel's approval, we gathered a group of friends and family who fell into our target demographic (Software Developers) and had them carry out the testing and complete the form. We also used Amazon Mechanical Turk, a crowdsourcing marketplace, to find additional survey respondents.

The form consists of 3 stages, first is a background information gathering stage where we ask users to anonymously fill out their experience and position in the software world. This was done to give us insights into potential links between certain software professionals and their viewpoints on the web application. The second stage is the testing phase where the user is asked to complete certain tasks in the application. The final stage is feedback where the user answers a series of questions about their experience testing the application.

6.4.1 Footnotes

1. Feedback Form: https://forms.gle/WVertvJDmwxtcJLN6
2. User Testing Results: Results

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.

6.5.1 Footnotes

1. Apache benchmark: https://httpd.apache.org/docs/2.4/programs/ab.html

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.