Ryerson University
Department of Electrical, Computer & Biomedical Engineering
Faculty of Engineering & Architectural Science

Job Recommender System

by

Polina Valiakhmetova, Karim Soubra, Jasdeep Gahunia, Harkirat Gill

Electrical & Computer Engineering Capstone Design Project

Toronto Metropolitan University, 2024

Acknowledgments

We would like to thank all of those who have contributed to the completion of our capstone project thus far. We extend our gratitude to our professor Alagan Anpalagan, whose guidance and support played a pivotal role in shaping the direction of our project. His valuable insights, constructive feedback, and unwavering encouragement significantly enriched the quality of our work.

Certification of Authorship

We, the undersigned, hereby certify that this report summarizing design activities titled "Job Recommender System" is the result of our original work. We declare that:

1. All contributions from each group member have been appropriately acknowledged within the document.
2. The content of this report is free from any form of plagiarism, and all external sources used in our research are duly cited and referenced.
3. We have not submitted this work, or any part thereof, for academic credit elsewhere.

Signatures

1. Name: Polina Valiakhmetova, Date: April 12th 2024
2. Name: Harkirat Gill, Date: April 12th 2024
3. Name: Jasdeep Gahunia, Date: April 12th 2024
4. Name: Karim Soubra, Date: April 12th 2024

Table of Contents

Abstract

The task of finding jobs that are tailored towards your interest and finding applicants that are appropriate for a job is a never-ending process that requires constant innovation since the task of matching a user to a job and vice versa is very subjective and information is constantly changing. The goal of the Job Recommender System is to simplify the process of matching applicants to jobs and vice versa by using a machine learning-driven approach. The main aspect of the Job recommender system is the machine learning model that uses a clustering algorithm and logistic regression to appropriately match users and jobs. Job posting websites and resume posting websites are used in order to gather the data required to train the machine learning model. Extracting the data from the websites required the use of web crawling techniques and natural language processing tools such as OpenAI's large language model. A Django backend was used alongside an interactive frontend in order to make the user experience as easy as possible when using the job recommender system. The results of the recommender system varied since the quality of results relies heavily on how much information the user has given the machine learning model, for example, a resume with no job experience and a minimal list of skills will have a harder time being matched with a job. Every recommendation had a confidence rating associated with it in order to show the user how much of a match an applicant or job posting is.

Keywords: large language model, recommender system, web crawling, Django, cluster, recommendation

Introduction & Background

Finding the perfect candidate or finding the perfect job is not a straightforward issue that many applicants and recruiters struggle with. The job recommender system uses a novel approach to solve this issue. The recommender system would help save the recruiter and applicant's time by only recommending jobs that are relevant to the applicant and recommending relevant applicants to recruiters. The recommender system is designed to be user-friendly and operate in the background, such that the application is simple and easy to use for recruiters and applicants. The recommender system is trained on data regularly in order to maintain its relevance to the user by recommending new jobs and new applicants, this also helps save time since the recruiter and applicant do not have to sift through many job postings and candidates. The quality of recommendations is also inspected by using a confidence rating in order to ensure users are receiving high-quality recommendations. The confidence rating will be covered in the measurements and testing procedures sections.

Objectives

The project's primary aim is to develop a Job Recommender System that deploys machine learning to enhance the recruitment landscape. This system will provide accurate recommendations to job seekers about open positions, and similarly, connect employers with well-suited candidates. This advanced system will harness up-to-date data from a wide range of professional platforms and job listings to produce personalized and appropriate recommendations. Our goal is to ensure that the job-matching process is not only relevant but also streamlined for maximum convenience and efficiency. To achieve this, the system will be built on flexible frameworks that support a modular and scalable architecture. Such a design will enable us to accommodate a growing user base and an ever-increasing and varied set of job listings without compromising on performance or speed. Emphasis will be placed on the system's maintainability, allowing for seamless updates and the smooth integration of new features and improvements in response to iterative user feedback and technological advancements. Moreover, we are dedicated to providing an outstanding user experience. The system's interface will be designed to be both intuitive and user-friendly, ensuring that users of all technical skill levels can navigate through various platform functionalities. Our ultimate goal is to develop an essential tool for job seekers to find roles that fit their qualifications and aspirations, and equally for employers to uncover the talent that meets their specific criteria. By successfully implementing this system, we anticipate stimulating a job market that is not just efficient but also dynamic and responsive to the evolving needs of the workforce and the economic landscape.

1.0 Theory and Design

1.1 WebApp Structuring and Schema

The theory behind the structuring of the backend revolves around the MVT (Model-View-Template) architecture utilized by Django. MVT allows for different components of a web application to be separated and handled in different files which in turn makes it easier to navigate and implement new additions to the application. The files break down into three main categories:

Models

These files define the structuring of the database tables and schema. Model classes are created that inherit Django's built-in model class and the attributes of these newly defined classes represent the tables columns or keys. By creating multiple tables, a database schema can be defined with foreign keys describing the relationships between the tables. For example, in our Job Recommender system, two separate models can be created for Resumes and Work Experience. From there, each Work Experience can have a foreign key that represents which Resume the Work Experience belongs to as seen below in Figure 1.1. This concept of database normalization allows the database to be highly scalable, perform better, and be much more flexible when dealing with resumes that may not provide every detail.

```
class Resume(models.Model):
    resumeID = models.CharField(max_length=255, unique=True, default='DefaultResumeID')
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True, related_name='resumes')
You, 1 minute ago | 1 author (You)
class WorkExperience(models.Model):
    experienceID = models.CharField(max_length=255, unique=True, default='DefaultExpID')
    resume = models.ForeignKey(Resume, on_delete=models.CASCADE, related_name='work_experiences')
```

Figure 1.1 Foreign Key Usage Work Experience to Resume

Views

View files define the interaction between the front end and Backend of an MVT-based web application. In our usage of the MVT architecture, each View is linked to a URL, which when accessed by a User in the front end, can respond to API requests. This is done as in a View file, View classes are defined and the methods of these classes can define the behavior of the backend based on whether a POST or GET request is sent by the FrontEnd. An example of our implementation of this can be seen in the SignUp View (Figure 1.2) which interacts with requests sent at "UserAuth/Signup/". This View's role is to facilitate the creation of a user after receiving a POST request from the FrontEnd at the previously specified URL with the necessary user data in the request.data attribute. Once the data is confirmed to be sufficient through the usage of a Django Form, the View passes the data over to a serializer which handles the creation of a new

user in the database. Once the View has handled the request, it can send an HTTP response to the front end based on the outcome of its logic.

```python
class SignUpView(APIView):
    def post(self, request):
        form = SignupForm(request.data)      Harkirat, 2 weeks ago • User Authentication Working
        if form.is_valid():
            serializer = UserSerializer(data=form.cleaned_data)
            if serializer.is_valid():
                serializer.save()
                return Response({
                    'success': True,
                    'message': 'Registration successful. Welcome to The Job Recommender System!'
                }, status=status.HTTP_201_CREATED)
            else:
                return Response({
                    'success': False,
                    'message': 'Registration failed. Please correct the following errors:',
                    'errors': serializer.errors
                }, status=status.HTTP_400_BAD_REQUEST)
        else:
            return Response({
                'success': False,
                'message': 'Invalid form data. Please correct the following errors:',
                'errors': form.errors
            }, status=status.HTTP_400_BAD_REQUEST)
```

Figure 1.2 Signup View

Frontend

In our web development project, we leveraged the power of JavaScript and TypeScript alongside React.js for crafting our frontend. This choice was made to capitalize on React's robust capabilities in creating highly interactive and responsive user interfaces. By utilizing JavaScript and TypeScript, we were able to write cleaner, more organized code with enhanced type safety, ensuring fewer bugs and smoother development workflows.

Our frontend architecture was structured around splitting the application into distinct pages and reusable components. This approach enabled us to efficiently manage complexity and promote code reusability throughout the project. Each page was designed as a standalone React component, encapsulating all the necessary UI elements and functionality specific to that page.

Moreover, we modularized our design by breaking down the UI into smaller, reusable components. This allowed us to abstract common UI elements such as headers, footers, buttons, and input fields into standalone components. By doing so, we could reuse these components across multiple pages, promoting consistency in design and functionality across the entire application.

The adoption of TypeScript further bolstered our development process by providing static typing capabilities. With TypeScript, we were able to catch type-related errors during compile time, reducing the likelihood of runtime errors and improving code quality. Additionally, TypeScript's

support for interfaces and type definitions facilitated better code documentation and improved collaboration among team members.

In terms of styling, we utilized CSS to define the visual presentation of our components. By separating styles from the component logic, we maintained a clean and maintainable codebase, making it easier to update and customize the UI as needed. CSS modules or styled components were employed to encapsulate styles within individual components, preventing style conflicts and promoting modularity.

1.2 Job and Resume Scraping Design

An important aspect of the job recommender system is the ability to obtain enough data to train the recommender model in order to give out accurate recommendations consistently, as well as to give the user a vast job and candidate pool in order to make sure that every user's needs are met whether it be an employer or an employee. The two main tools used to implement the design are NLP models and web drivers.

Resume Scraping Pipeline Design

The resume scraping pipeline in Figure 1.2.1 has four stages and ends with the resume user data that is ready to be used in the recommender system. The first stage of the pipeline is to open and read the given pdf file. The second stage is to clean the text before parsing, this stage consists of removing random uninterpretable symbols such as bullet points that are commonly found on resumes when listing items. The third stage mainly consists of simple text processing consisting of using regular expressions (regex) which is a Python text processing library that helps with the more complex text searches, regex enables text processing by defining the text as a pattern with predefined symbols[1]. The third stage is mainly used to extract data that is relatively easy to parse such as email, web links, and phone numbers. The final stage is the main stage where the more complex text processing takes place.

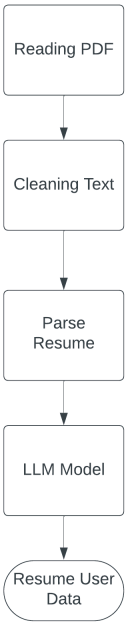

Figure 1.2.1 Resume scraping pipeline

The final stage of the resume scraping pipeline uses OpenAI's large language model. The final stage uses OpenAI's Large language model (LLM) in order to interpret the text in the resume and extract the information that is needed. The LLM does this by using a pre-trained transformer also known as GPT. The LLM can extract data by converting the blocks of text in the resumes into tokens, with each word equating to a token, then converting the tokens into word embeddings which is a vector representation of the word, this conversion can then help the large language model to interpret the text and understand the meaning of the blocks of text[2]. After passing the data to the LLM, the model then extracts the user's information such as job experience, job titles, and education.

Job Scraping Pipeline Design

Figure 1.2.2 shows the job scraping pipeline has 4 main stages and returns a job posting data set that is ready to be used in the recommender system. The first stage uses Selenium to emulate a web browser such as Chrome. This is done using a web driver which automates web browsers and enables code to interact with the web[3]. The second stage of the pipeline is to interact with the via the web driver, this is a key step since the job postings are not always fully displayed on the website to scrape, therefore there is a need to navigate to the desired part of the site that is to be scraped by the web scraper. The third stage of the pipeline extracts the data from the website, which in this case would be the job post. The fourth stage cleans the text extracted from the job posting, if necessary, and then extracts the rest of the information required from the post. Finally, this results in a dataset that is ready to use by the recommender system. The fourth

stage also uses the same LLM as the resume scraping pipeline, except for the final stage of the job scraping pipeline, the main data extracted is the list of skills a job posting has listed.

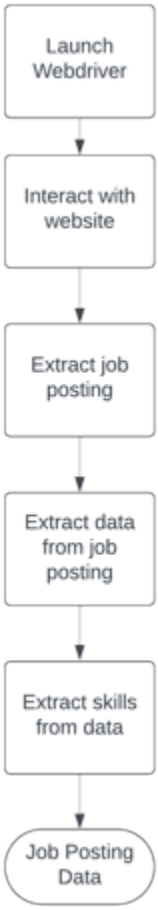

Figure 1.2.2 Job scraping pipeline

1.3 Machine Learning Algorithm Design

The machine learning algorithm for the recommender system consists of a few different components that work together to provide personalized recommendations to users.

Data Preparation

The first step before training the model is to clean and vectorize the data, then transform it into matrices. This involves removing unnecessary symbols, punctuation, and specific words that do not contribute to the model's understanding of the data. The cleaning process is tailored to preserve the meaningful content while eliminating noise that could detract from the model's accuracy. After cleaning, the text data (job descriptions, skills, titles) is converted into numerical vectors which enable mathematical operations. To achieve this TD-IDF (Term Frequency-Inverse

Document Frequency) is used which quantifies the importance of a word in a document relative to a set of texts. The TF-IDF score increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the text[4]. The formula for TF-IDF is shown in Equation 3 where TF is the term frequency and IDF is the inverse documents frequency.

$$TF = \frac{frequency\ of\ term\ in\ the\ document}{total\ number\ of\ terms\ in\ document}$$

Equation 1: Term Frequency (TF)

$$IDF = log(\frac{number\ of\ documents\ in\ corpus}{number\ of\ documents\ in\ corpus\ containing\ the\ term})$$

Equation 2: Inverse Document Frequency (IDF)

$$TF\text{-}IDF = TF * IDF$$

Equation 3: TF-IDF equation

Each field is treated distinctly because they carry different weights and meanings in the context of job recommendation. For instance, specific skills may be more predictive of job relevance than the broader descriptions provided in job postings. Job titles are vectorized with a broader n-gram range to capture combinations of words that often represent unique jobs. Skills are vectorized with a focus on capturing both individual skills and their combinations, reflecting the detailed qualifications required for a job. Descriptions, being the most verbose, are processed with a focus on extracting meaningful terms while excluding common job-related words and stop words. The resulting matrices from vectorizing job titles, skills, and descriptions are then combined into a single matrix. This combined matrix serves as the input for clustering algorithms, which group similar jobs and applicants to facilitate accurate recommendations.

K means Clustering

Once all the data is prepared the jobs are grouped based on their features into clusters such that items within a cluster are more similar to each other than to those in other clusters. The chosen clustering method is K-means which aims to group *n* observations into *x* clusters in which each observation belongs to the cluster with the nearest mean. It starts by selecting k initial centroids and then assigning each job listing to the nearest centroid. The Euclidean distance between the point and the centroid is calculated to find the nearest points[6]. Once all the points are assigned a cluster, the centroids are recalculated until the centroids no longer change significantly.

To find the optimal number of clusters 2 different methods were used. The first one is the elbow method which uses a graphical approach, as seen in Figures 1.3.1 and 1.3.2, to identify the point where the within-cluster sum of squared distances (WCSS) starts declining at a slower

rate[6]. The equation of WCSS is shown below where C_i is the set of all points in the cluster i and u_i is the centroid of C_i.

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2$$

Equation 4: Within-Cluster Sum of Squared Distances

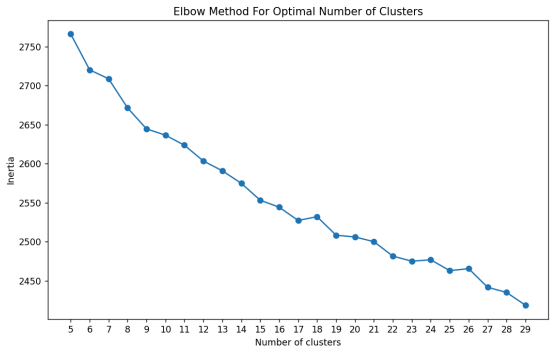

Figure 1.3.1: Plot of Accuracy vs Number of Clusters for Applicants

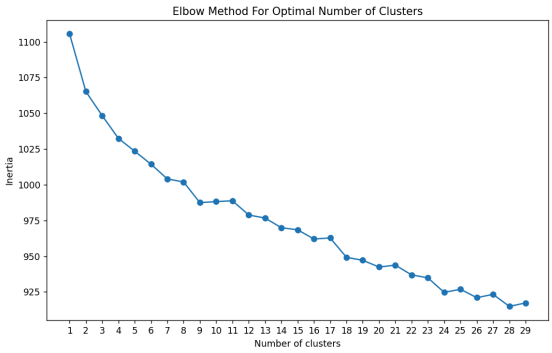

Figure 1.3.2: Plot of Accuracy vs Number of Clusters for Recruiters

In the analysis, two figures were generated using the elbow method to evaluate the clustering behavior of our dataset. However, in both instances, a clear "elbow" point—a definitive indicator of the optimal cluster count—was not present. The absence of a pronounced elbow complicates the task of selecting an appropriate number of clusters purely based on this criterion. Given this ambiguity in the elbow plots, an alternative approach for determining the optimal number of clusters was used.

The second method was a silhouette score, which measures the degree of separation between clusters by calculating how similar an object is to its cluster compared to other clusters. The score ranges from -1 to 1, where a high value indicates that the object is well-matched to its cluster and poorly matched to neighboring clusters. In essence, a high silhouette score signifies that the clusters are distinct and well separated from each other, with objects being closely related to their cluster. This method provides a quantitative basis to assess the appropriateness of the number of clusters by considering both the cohesion within clusters and the separation between them. Utilizing the silhouette score allows for a more nuanced evaluation of clustering performance, especially in cases where the elbow method does not yield a clear-cut result.

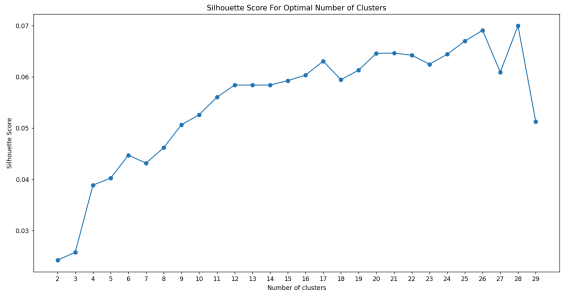

Figure 1.3.3: Plot of Accuracy vs Number of Clusters for Applicants

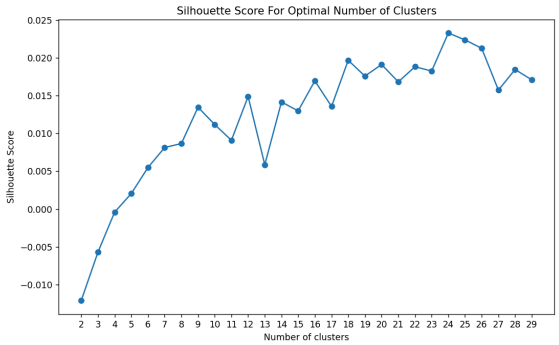

Figure 1.3.4: Plot of Accuracy vs Number of Clusters for Recruiters

Analyzing the two preceding plots, we determined that the optimal number of clusters for the applicant recommender system is precisely 20. This decision was informed by observing that beyond this threshold, the data's stability diminishes, indicating a less coherent clustering structure. For the recruiter model, the ideal clustering range was identified as between 7 and 8 clusters, balancing granularity with meaningful separation. Following the selection of these cluster counts, we employed 3D plotting to validate our choices. This visualization confirmed that the clusters were indeed well-formed and cohesive, rather than being dispersed or overly diffuse, underscoring the effectiveness of our clustering strategy.

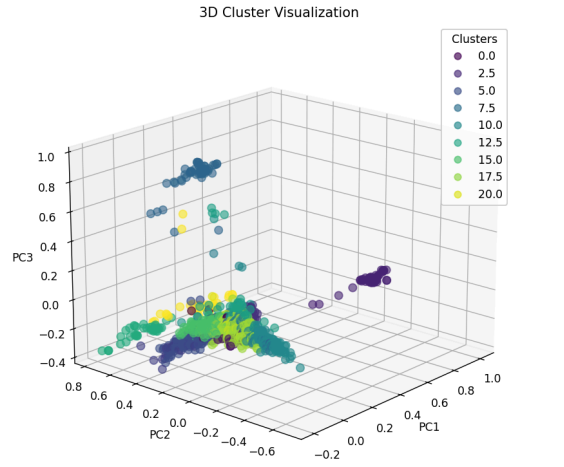

Figure 1.3.5: Plot of 20 Clusters for Applicants

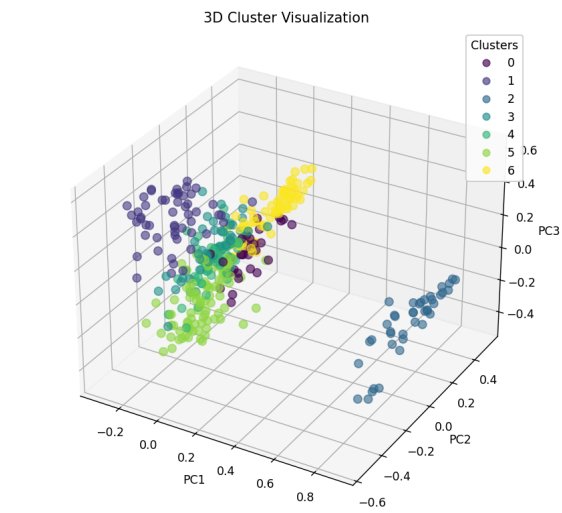

Figure 1.3.6: Plot of 7 Clusters for Recruiters

Cluster Logistic Regression

After completing the clustering phase, logistic regression was used to predict the likelihood that a given job listing belongs to one of the clusters and assign that job a cluster. The logistic regression model uses the logistic function, also called the sigmoid function, to squeeze the output of a linear equation between 0 and 1. Equation 5 represents the logistic regression equation that calculates the probability of the output being 1 where, β_0, $\beta_1 \ldots \beta_n$ are the model's coefficients and, $X_1, X_2 \ldots X_n$ are the features[7].

$$h\Theta(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_1 X_1)}}$$

Equation 5: Logistic Regression

Once all the jobs/applicants are assigned a cluster, the cluster table in the database is updated so this information can be used later on to provide recommendations.

Feedback Logistic Regression

Lastly, a second logistic regression was added to predict the likeness of a job posting based on user feedback. Incorporating user feedback involves adjusting the weights or importance of items based on user interactions. For instance, if a user likes (thumbs up) an item, the weight for similar items increases, and vice versa for dislikes (thumbs down). The weights for the project are set to 1 being a job liked and -1 for a disliked job. Once all the data is collected, jobs with several ratings are summed up to ensure each job has one score. Next, the feedback is set as the target variable in the regression function, allowing the model to learn to associate specific patterns in the job features with the likelihood of receiving positive ratings.

This mechanism allows the recommendation system to adapt dynamically to users' preferences every time new feedback is added. Over time, as more feedback is accumulated, the system's recommendations can become increasingly personalized, closely aligning with user preferences and improving accuracy. The interplay between content-based features, collaborative filtering elements, and dynamic feedback creates a robust system capable of adapting to user preferences and discovering new items likely to interest the user.

Recommendation Function

The final recommendation for a user is created by first using a previously trained logistic regression model that predicts the cluster to which a user belongs. Then, the cosine similarity is calculated between the users' information and the jobs within the predicted cluster. This metric measures the cosine of the angle between two vectors as seen in Equation 7, serving as a measure of similarity between them[8].

$$similarity(A, B) = cos(\theta) = \frac{A \bullet B}{||A||\,||B||}$$

Equation 7: Cosine similarity

The output of the cosine similarity are scores for each job which are saved in descending order. The top 30 of these recommendations are stored in a database table which allows for fast recommendations after the initial set. Next, the second logistic regression model is used, which calculates the probability of a user liking the job. The 2 scores are then normalized respectively so that a confidence rating can be calculated. In Equation 8 the 2 normalized scores are added together with different weightings to produce a final confidence score. Since the feedback data is limited, the feedback score is weighted less than the cluster score.

$$Confidence\ rating = \alpha S_{cluster} + \beta S_{feedback}$$

$$\alpha = 0.7,\ \beta = 0.3$$

$$S_{cluster} = normalized\ score\ for\ cluster\ logistic\ regression$$
$$S_{feedback} = normalized\ score\ for\ feedback\ logistic\ regression$$

Equation 8: Feedback Weight Calculation

2.0 Alternative Designs

2.1 WebApp Structuring and Schema

For the web app structuring and schema, our group chose to go with a high level of normalization. This meant that every time we would initiate a search for an attribute such as the skills of a resume, we would need to first locate the Resume ID of that specific resume and then gather all the skills related to that Resume ID through another traversal of the skills table. This structuring was theoretically clearer as it decoupled a lot of the attributes of a Resume or Job Posting, but we found that this not only increased querying times but also made it a lot harder to keep track of all the different components of a single model. An alternative to this form of structuring would be to keep all the primitive forms of data localized within the Resume and Job Posting models rather than having them related in different models. This limits the amount of querying needed to be done as once the Resume ID is located, the query is essentially completed as the list of skills is present within that entry. While this form of structuring would/did save time in querying, it is simply not possible for many of the attributes within the Job Posting such as Work Experience or Education. This is because these attributes have multiple attributes within themselves such as the dates, titles, and degree names.etc associated with them. Having them as a single attribute within the Job Posting or Resume table would limit them to simply being a list of Job Titles or Degrees for each form of education. While some areas of the code were adjusted to use the correct form of structuring, much of the code that had already been completed and was deemed working was not changed to preserve the working state.

Another alternative design in the schema of the database would have been to limit the number of unique skills in the database by hard coding or filtering for a select group of skills. This would have been beneficial as our current design allows for any skill to be added to our skills table so long that it is unique. This leads to a high number of useless/redundant skills that do not benefit the clustering done by the machine learning algorithm. With a limited number of skills, we could ensure that each skill attributed to a user is correctly weighted in the algorithm and also make it a lot simpler for both the front end and back end to handle a list of skills.

```python
class ListOfSkills(models.Model):

    skill_name = models.CharField(max_length=200, default='Default Skill')

    def __str__(self):
        return self.skill_name

class JobPosting(models.Model):
    #jobID = models.AutoField(max_length=255, unique=True)
    title = models.CharField(max_length=200, null=True, blank=True)
    company_name = models.CharField(max_length=200, null=True, blank=True)
    location = models.CharField(max_length=200, null=True, blank=True)
    job_description = models.TextField(null=True, blank=True)
    posted_date = models.DateField(default=timezone.now)
    application_deadline = models.DateField(default=timezone.now)
    experience_required = models.CharField(max_length=255, null=True, blank=True, default='No experience required')
    benefits = models.TextField(max_length=700, null=True, blank=True)
    employment_type = models.CharField(max_length=200, null=True, blank=True)
    skills = models.ManyToManyField(ListOfSkills, related_name='job_postings', blank=True)
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True, related_name='creator')
```

Figure 2.1.1 Job Posting Model with attribute being built from referenced Model

```
class JobPosting(models.Model):
    #jobID = models.AutoField(max_length=255, unique=True)
    title = models.CharField(max_length=200, null=True, blank=True)
    company_name = models.CharField(max_length=200, null=True, blank=True)
    location = models.CharField(max_length=200, null=True, blank=True)
    job_description = models.TextField(null=True, blank=True)
    posted_date = models.DateField(default=timezone.now)
    application_deadline = models.DateField(default=timezone.now)
    experience_required = models.CharField(max_length=255, null=True, blank=True, default='No experience required')
    benefits = models.TextField(max_length=700, null=True, blank=True)
    employment_type = models.CharField(max_length=200, null=True, blank=True)
    skills = models.ManyToManyField(ListOfSkills, related_name='job_postings', blank=True)
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True, related_name='creator')

    skills_list = models.JSONField(default=list)

    cluster = models.IntegerField(max_length=255, default=0)
```

Figure 2.1.1 Alternative Design with skills_list localized within Job Posting

2.2 Job and Resume Scraping Design

The alternative designs for the resume and job scraping pipeline involved examining designs that are more efficient at data processing and data extraction. The major drawback of such alternatives is that the complexity of the pipelining increases drastically. Figure 2.2.1 is the resume scraping pipeline. This alternative design shares some design similarities with Figure 1.2.1. The first two stages of the pipeline are similar to Figure 1.2.1. The pipeline in Figure 2.2.1 starts with reading the resume and then cleaning the text, the next stage however is where the new design begins both the Parse resume and LLM model would be running in parallel. In Figure 1.2.1 the pipeline is executed sequentially where parsing the resume would start and finish, and then feed the LLM the rest of the resume, this approach's main advantage is its simplicity as running it sequentially would not lead to any concurrency errors, and implementing the design is relatively easy, but comes with the drawback that the pipeline may be slow since everything is running sequentially any point of the pipeline can serve as the bottleneck. The alternative design in Figure 2.2.1 executes the parse resume stage and the LLM model in parallel, the main benefit is that the two critical stages of the data extraction process are executed independently and in parallel with each other therefore speeding up the resume scraping process, this would require multithreading and implementation of appropriate concurrency mechanisms in order to make sure both threads are in sync and critical sections of the code are respected. The advantage of the alternative design is its speed however the trade-off would be the added complexity of implementing the pipeline.

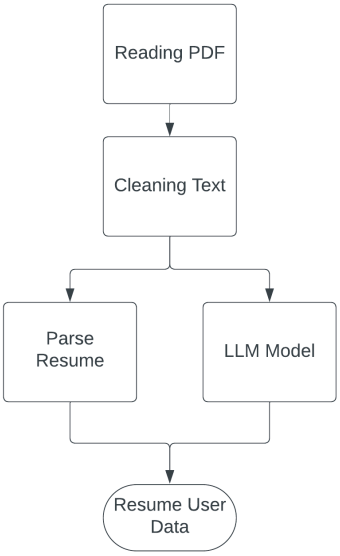

Figure 2.2.1 Alternative resume scraping pipeline

The alternative job scraping pipeline in Figure 2.2.2 is similar to the pipeline shown in Figure 1.2.2. The first three stages of the alternative job scraping pipeline are the same as the pipeline shown in Figure 1.2.2 These stages cannot be changed since the first three stages are associated with interacting with the job posting website to extract the job posting. These stages cannot operate in parallel. The main design changes in Figure 2.2.2 are with the data extraction of a job post. The design in Figure 1.2.2 is executed sequentially, first extracting data from the job posting and then extracting the skills in the job post using an LLM. The alternative design on the other hand executes the data extraction stages in parallel. Executing the data extraction in parallel speeds up the pipeline since the data extraction stages are the slowest stages in the pipeline. The main drawback of executing the data extraction stages in parallel is the added complexity of running code in parallel. Measures must be taken in order to ensure synchronization between threads executing in parallel.

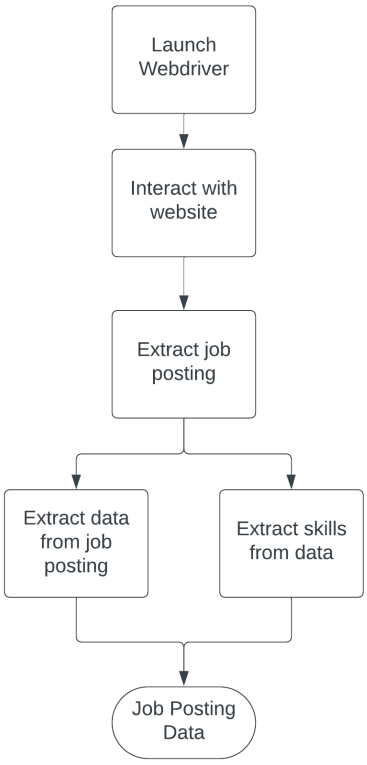

Figure 2.2.2 Alternative Job scraping pipeline

2.3 Machine Learning Algorithm Design

In exploring alternative designs for our hybrid model job recommender, we considered a Deep learning approach as a potential solution. Similar to the TF-IDF vectorization in the hybrid model, deep learning models can automatically learn representations from raw text data. Using embeddings such as Word2Vec, GloVe, or contextual embeddings from models like BERT, this approach captures semantic relationships between words, potentially offering richer feature representations than TF-IDF. Next, a neural network is an option for the model architecture as it is well-suited for learning complex patterns in the sequence of words, capturing the context and the semantic meaning more effectively. Deep learning models can be trained to predict user-job interactions directly, leveraging user feedback, job descriptions, and user profiles in an end-to-end manner. This could eliminate the need for separate clustering and classification stages, offering a more integrated approach to personalization.

Comparing the performance of the hybrid model to a deep learning approach, the deep learning models, particularly those based on transformers, have shown superior performance on a range of natural language processing tasks compared to traditional machine learning approaches. They could potentially offer more accurate recommendations by better capturing the nuances of job descriptions and user preferences. Additionally, deep learning models are highly scalable and can handle larger datasets more effectively as computational resources increase. However, the hybrid model offers more flexibility in adjusting and interpreting individual components of the recommendation pipeline.

3.0 Material/Component List

3.1 Component List

The job recommender system has 4 main sections: frontend, backend, machine learning model, and web scraping. In the front end the main component used is React, React is a library that allows the design of a User interface and enables the front end of the web application to connect to the backend, where all the logic of the web application occurs. The backend is where the logic for the frontend is implemented as well as where the database is designed and managed. The main components of the backend are Django and Amazon Web Services. Django is used to implement the backend and Amazon Web Services is used to host the database. Django enables rapid development of the backend, which enables the implementation of features much quicker and simpler[9], and Amazon Web Services enables the deployment of a relational database that can be used to store our user's data to train and use our machine learning models[10]. The web scraping section of the job recommender system is tasked with data collection and cleaning in order to be fed into the machine learning model. The components of the web scraping section of the recommender system are Beautiful Soup 4 and Selenium, for web scraping and web crawling, and OpenAI API for text analysis. The final part of the job recommender system is the machine learning model, the machine learning model utilizes various machine learning techniques to provide accurate recommendations. Sklearn is used to implement the machine learning techniques used and joblib is used to save the machine learning model. Pandas and Numpy are used throughout the project as a way to efficiently manipulate data. Finally, the project is stored on a GitHub repository.

Components:

- Beautiful Soup 4
- Selenium
- OpenAI API
- Amazon Web Services
- Django
- React
- NLTK
- Sklearn
- joblib
- Numpy
- Pandas
- Github

3.2 Pricing

All the costs associated with implementing the job recommender system follow a pay-as-you-go model, with a free beginner tier and higher tiers that cost more as you go through the tiers. The tiers increase as the usage of the recommender system increases. Amazon Web services have a whole range of relational databases that can be used depending on the application, the database options include databases that are optimized for general purpose,

memory, compute, burstable, and reads[11]. Each type of database has different costs associated with them and is more costly the more they are used. The database used in the job recommender system is a general-purpose database from Amazon Web Services, this option was chosen as it provides optimal performance in all aspects and is used for more general-purpose applications, which is denoted by db.mX in Amazon Web Services [11]. The Amazon Web services pricing calculator was used to determine the approximate cost of running a db.mX server since the pricing model of the database is pay-as-you-go, the prices may vary depending on the utilization of the server[12]. Figure 3.2.1 displays the current estimated cost of running the database. The estimates were calculated based on using the db.m4.xlarge and utilization of about 50% per month. The monthly cost would be 303.90 USD. However, the price can fluctuate depending on the utilization as a result the monthly cost can be as low as 53.51 USD per month and as high as 559.40 USD per month.

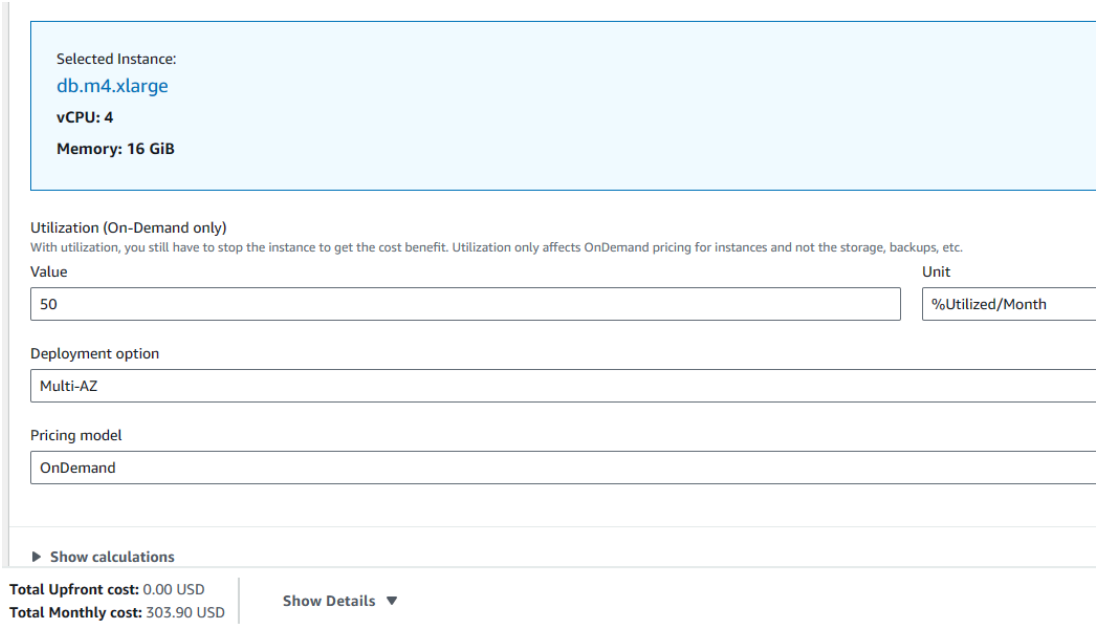

Figure 3.2.1 Price calculation for database

The OpenAI API used for text analysis and data extraction from resumes and job postings is also on a pay-as-you-go basis. OpenAI's pricing model is also pay-as-you-go and is calculated per token, with a token representing approximately 4 letters for example an 8-letter word is approximately 2 tokens[13]. Figure 3.2.2 shows the pricing of the large language model used for text analysis. Currently, the price of sending data to the language model is 50 cents per million tokens and the price for receiving data from the language model is about 1.5 USD per million tokens[13]. Therefore, the cost of using OpenAI correlates proportionally to how much of OpenAI's API is used. The current estimate for scraping about 400 resumes and 1000 jobs per month will cost about 30 USD per month. The pricing may also increase if more complex language models are used. As seen in Figure 3.2.3 the price for using GPT4 is about 30 USD per million tokens for inputs and 60 USD per million tokens for outputs[13].

Model	Input	Output
gpt-3.5-turbo-0125	$0.50 / 1M tokens	$1.50 / 1M tokens
gpt-3.5-turbo-instruct	$1.50 / 1M tokens	$2.00 / 1M tokens

Figure 3.2.2. OpenAI gpt 3.5 pricing.

Model	Input	Output
gpt-4	$30.00 / 1M tokens	$60.00 / 1M tokens
gpt-4-32k	$60.00 / 1M tokens	$120.00 / 1M tokens

Figure 3.2.3. OpenAI gpt 4 pricing.

Github is used to store the code in a repository, this acts as a safe space for development and a safe space to store code. The GitHub repository also stores the machine learning model that is generated, as a result, Github's large file storage is used. The cost of using Github large file storage comes down to the number of data packs used, as github defines a data pack as 50 GiB of storage and each data pack costs 5 USD per month[14]. The cost of using GitHub large file storage is 5$ per month since only one data pack is used. Figure 3.2.x4 shows an example of how much would one data pack cost for a whole year.

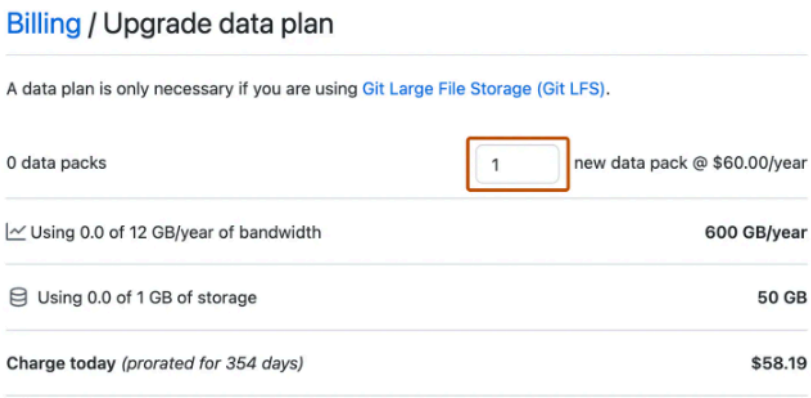

Figure 3.2.x4 Github large file storage yearly cost for one data pack.

4.0 Measurement and Testing Procedures

After completing the core functionality outline for our project, our team began testing the Job Recommender System to ensure that it was working correctly. There were three main areas of testing that we conducted, functionality testing, performance testing, and recommendation testing.

The first area, functionality testing was done to ensure that all of the different features added to the web application were working correctly and that there were no edge cases in user input that could alter the expected results from the actual results. To conduct this testing, each of the group members went through the complete user workflow on both the applicant and recruiter sides, ensuring that every feature developed was working as intended. On the applicant side, this meant first creating a user, uploading a resume, retrieving the recommendations generated, liking or disliking the recommendations, searching for a job posting, applying to a job posting, viewing liked jobs and finally editing the user profile. This was done in conjunction with the recruiter side where a similar workflow was simulated but rather than getting recommended jobs, each job posting was recommended candidates. While completing this testing, our group also made sure to track whether job postings were being accurately displayed to users based on their resumes. A test that we conducted was to create an applicant with a list of skills and experience that we can call “X”. We then created a job posting on the recruiter side that was highly similar to the list of skills and experience “X”. From there, we checked that both the applicant was getting recommended for the job posting and that the recruiter was getting recommended to the applicant for that specific posting. This fulfilled the base truth criteria that we were looking for when working with the machine learning algorithm. More comprehensive measurements and testing were also conducted for the algorithm using a scoring method in the recommendation testing.

The next area of testing that we conducted was performance testing. This testing was specifically for the performance of backend views when a request by the user. We decided to perform this testing as we noticed that both our recommendations and searching features on both the recruiter and applicant side were slowing down as we increased the number of entries within our database. Before beginning testing, we decided to stop adding additional entries to the database so that we could get a benchmarked score and compare against this score in the future while maintaining the variable of entries. From there, we made multiple changes to how querying our database was done and also how the database was structured to improve the performance of both the recommendation and searching views. The results of our testing and an analysis of those results are present in 5.0 Performance Measurement Results and 6.0 Analysis of Performance

The final area of testing was the recommendation and model testing. This consisted of using the accuracy_score and classification_report from the sklearn.metrics module are used to evaluate the performance of logistic regression models. These metrics serve different purposes in understanding how well the model performs in classifying the test data. The accuracy_score function calculates the proportion of correct predictions over the total number of predictions made. To do so the data was split into testing and training sets, which would be used to train the model and then verify the results.

The classification_report function provides a more detailed analysis, breaking down the performance of the model for each class or category being predicted. The precision metric is a measure of the model's accuracy in predicting positive instances for that class. In other words, of all the instances the model predicted for a class, how many were actually correct? Next, the recall metric measures the model's ability to detect positive instances. Similarly, of all the actual instances of a class, how many did the model correctly identify? The F1-score is another metric in the classification_report which is a mean of the precision and recall, providing a single metric to assess the balance between them. The last metric is the support which is the number of actual occurrences of the class in the specified dataset, providing insight into the dataset's balance. This breakdown allows us to understand the model's strengths and weaknesses, especially in a dataset where some classes are more represented than others.

5.0 Performance Measurement Results

After completing both the benchmark and subsequent testing after making changes to the querying and database structuring, we were presented with the following results. On the applicant side, the search view went from a 7-second delay to a 2-second delay and the recommendation view went from a 47-second delay to a 6-second delay. On the recruiter side, the search view went from a 7-second delay to a 4-second delay and the recommendation view went from a 16-second delay to an 8-second delay.

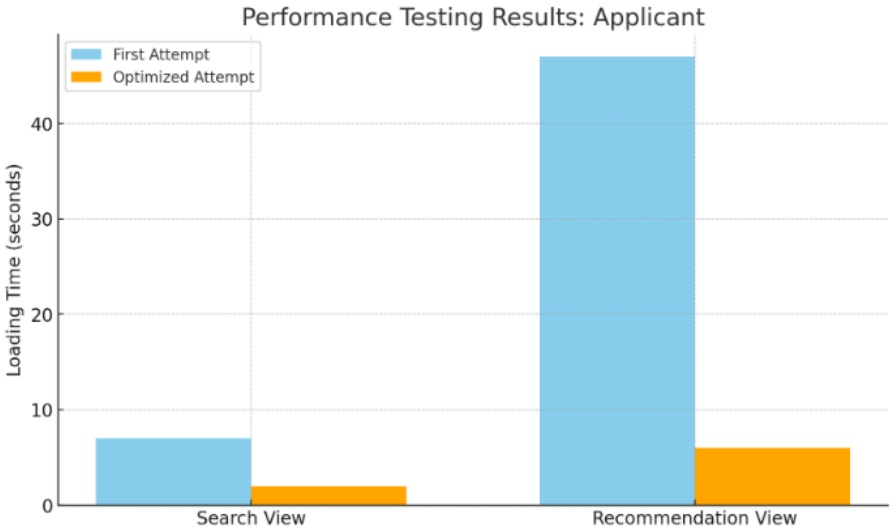

Figure 5.1 Initial Delays vs Optimized Delays Applicant Side

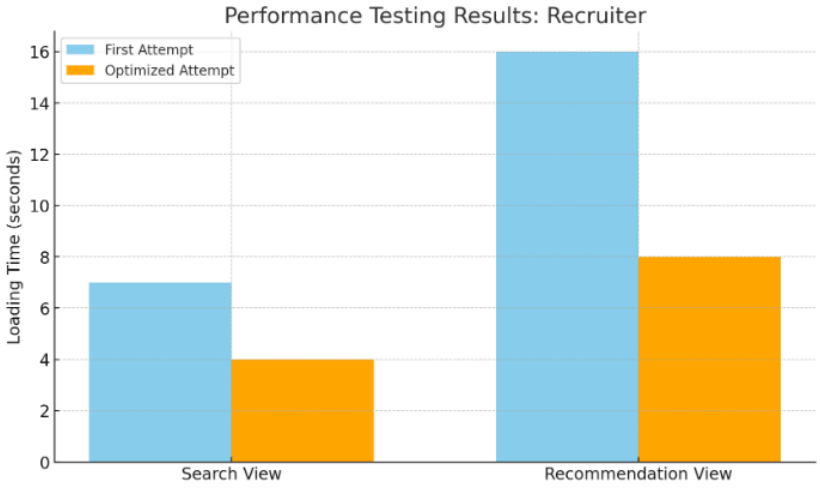

Figure 5.2 Initial Delays vs Optimized Delays Recruiter Side

For the recommender system, the accuracy_score and classification_report were calculated for both the applicant and recruiter sides and for the 2 logistic regression instances. Figure 5.3 shows the results for the applicant model which recommends jobs to applicants, for the first logistic regression, that predicts clusters for jobs. The “Accuracy”, which is the output of

the accuracy_score, is 92% and the rest of the information in Figure 5.3 is the breakdown provided by the classification_report. Figure 5.4 shows the results for the second logistic regression model which predicts the likelihood of a user liking a job based on the feedback. Similarly, the "Accuracy for probability" is the output of the accuracy_score and the rest of the information is the classification_report breakdown.

Accuracy: 0.9233449477351916

	precision	recall	f1-score	support
0	1.00	0.67	0.80	6
1	1.00	1.00	1.00	13
2	1.00	0.86	0.92	14
3	0.00	0.00	0.00	2
4	1.00	0.92	0.96	13
5	1.00	1.00	1.00	9
6	1.00	1.00	1.00	7
7	0.91	0.71	0.80	14
8	1.00	0.67	0.80	3
9	1.00	0.33	0.50	3
10	1.00	0.60	0.75	5
11	0.90	1.00	0.95	37
12	0.93	0.93	0.93	14
13	0.94	1.00	0.97	16
14	1.00	1.00	1.00	13
15	0.80	0.86	0.83	14
16	1.00	1.00	1.00	11
17	0.85	0.99	0.91	70
18	1.00	0.95	0.98	21
19	1.00	0.50	0.67	2
accuracy			0.92	287
macro avg	0.92	0.80	0.84	287
weighted avg	0.92	0.92	0.92	287

Figure 5.3 Initial Delays vs Optimized Delays Recruiter Side

Accuracy for probability: 0.9790940766550522

	precision	recall	f1-score	support
-1	0.00	0.00	0.00	1
0	0.98	1.00	0.99	281
1	0.00	0.00	0.00	5
accuracy			0.98	287
macro avg	0.33	0.33	0.33	287
weighted avg	0.96	0.98	0.97	287

Figure 5.4 Initial Delays vs Optimized Delays Recruiter Side

As described for the applicant side model, the recruiter model results are presented in a similar manner. Figure 5.5 shows the results of the accuracy_score as "Accuracy" for the clustering logistic regression, which is 90%. The classification_report can be found underneath the accuracy results in the same Figure. For the feedback-based logistic regression, Figure 5.6 portrays the results for accuracy_score labeled as "Accuracy for probability" which is 95%. Similarly, the classification breakdown can be found in the same image, underneath the accuracy_score results.

Accuracy: 0.9047619047619048

	precision	recall	f1-score	support
0	1.00	0.33	0.50	3
1	1.00	1.00	1.00	9
3	0.83	0.71	0.77	7
4	1.00	1.00	1.00	4
5	0.86	0.98	0.91	43
6	1.00	0.83	0.91	18
accuracy			0.90	84
macro avg	0.95	0.81	0.85	84
weighted avg	0.91	0.90	0.90	84

Figure 5.5 Initial Delays vs Optimized Delays Recruiter Side

Accuracy for probability: 0.9523809523809523

	precision	recall	f1-score	support
-1	0.00	0.00	0.00	1
0	0.95	1.00	0.98	80
1	0.00	0.00	0.00	3
accuracy			0.95	84
macro avg	0.32	0.33	0.33	84
weighted avg	0.91	0.95	0.93	84

Figure 5.6 Initial Delays vs Optimized Delays Recruiter Side

6.0 Analysis of Performance

By analyzing the results of the delay-based performance testing, we can see that by localizing many of the attributes relating to both Job Postings and User Resumes, we can decrease the querying time therefore decreasing the delay in view completion. Observing the applicant side specifically, we saw a 71% decrease in search times and an 87% decrease in recommendation times as well. On the recruiter side, we saw a 43% decrease in search times and a 50% decrease in recommendation times. These results indicate that our hypothesis of localizing certain primitive attributes to the Job Posting and Resume tables rather than having them in their separate tables resulted in a decrease in query times. As the majority of the processing delay within a view is simply the time it takes to query the database, this resulted in a drastic decrease in delay when sending a request to the view. We could not completely remove all the delay as there were elements that could not be optimized such as a search through a single table based on an attribute of that table/model. Similarly, for the recommendation, we could not avoid running the machine learning algorithm whenever sending a request to the recommendation view.

The evaluation of the recommender system's logistic regression models reveals that the clustering model for the applicant recommendation system boasts a high overall accuracy of 92.3%, indicating a strong capacity for job categorization. Most clusters have perfect precision scores, signaling a high likelihood of correct job assignment predictions. Nonetheless, certain clusters displayed lower precision, pointing to occasional misassignments. The recall varies more significantly, with several clusters demonstrating high recall, thus capturing the most relevant jobs, while others have lower recall, indicating missed assignments. F1 scores, reflecting the balance of precision and recall, are generally high, although some clusters with lower scores suggest potential areas for improvement. Support levels vary, with less-populated clusters presenting less reliability in their precision and recall, potentially impacting the confidence in cluster assignment for these groups.

The feedback logistic regression model exhibits a remarkable overall accuracy of 98% in predicting user preferences for jobs. This high accuracy is predominantly due to its exceptional performance in predicting the neutral class '0', with near-perfect precision and recall. However, the model shows limitations in identifying the positive '1' and negative '-1' feedback classes, as evidenced by a precision and recall of 0.00 for these categories. This discrepancy is likely due to the substantial class imbalance, as the overwhelming majority of data points belong to the neutral class. While the weighted average metrics appear high, reflecting the dominance of the '0' class, the macro averages suggest room for improvement which could be solved by collecting more user feedback.

The recruiter recommendation system's cluster logistic regression model showcases an overall accuracy of 90.5%, reflecting a robust capability in accurately grouping potential applicants. The precision across the clusters is largely high, with some clusters achieving perfect scores, which suggests that the model is quite reliable when it assigns applicants to those clusters. Nonetheless, there are variations in recall, indicating that while some clusters are identified with high accuracy, others may be overlooked or misclassified, hinting at areas where the model's sensitivity could be improved. The F1 scores corroborate these findings, being high for certain clusters and indicating a balance between precision and recall, yet lower for others,

which may guide future model refinements. The support count shows the disparity in cluster sizes, which could influence the reliability of the model's performance metrics across different clusters.

In contrast, the feedback logistic regression model for the same system demonstrates an exceptional overall accuracy of 95.2% when predicting the likelihood of a recruiter favoring an applicant. This impressive figure primarily arises from the model's ability to predict neutral feedback with substantial precision and recall. However, the model needs to recognize positive and negative feedback, indicated by zero precision and recall for these classes, suggesting an inability to identify strong preferences. This challenge is underscored by the imbalance in the dataset, where the neutral feedback significantly outweighs other types. Although the weighted averages are high, they are driven by the model's performance on the dominant neutral class, and the low macro averages reflect the need for enhanced data diversity, especially for underrepresented feedback classes.

Conclusions

Summary of Completion

The Job Recommender System project culminated in the successful development of a sophisticated platform designed to bridge the gap between job seekers and recruiters. Utilizing a machine learning-driven approach, the system harnessed the power of Django for the backend to manage data processing, user authentication, and server-side logic efficiently. The frontend was developed using React, providing a responsive and intuitive user interface that facilitated seamless interaction between the system and its users. This combination ensured a robust infrastructure capable of handling complex functionalities like web scraping for data collection, natural language processing for insightful analysis of resumes and job postings, and the application of machine learning algorithms for generating accurate and personalized job recommendations. Despite challenges, the project team delivered a fully functional system that significantly advances the job-matching process.

Discrepancy Between Objectives and Achievements

The project aimed to develop an accessible, scalable online system that could adapt to the dynamic nature of the job market. Although the system achieved substantial functionalities, the objective of hosting it online was not met due to financial constraints, highlighting a significant gap between the project's ambitions and its practical outcomes. This aspect underscores the importance of budgetary planning in web application development, especially concerning operational costs like database hosting and API usage.

Unresolved Major Difficulties

Key challenges that remained unresolved included optimizing database performance due to high normalization levels and addressing bottlenecks in sequential data processing pipelines. Additionally, the project did not accomplish the hosting of the website, a primary objective, primarily because of the high costs associated with maintaining an online platform. These challenges indicate areas where future work is critically needed to enhance the system's performance and accessibility.

Future Work

Future enhancements for the Job Recommender System would focus on optimizing the database schema, implementing parallel processing to enhance data handling efficiency, and exploring more affordable solutions for web hosting. Efforts would also be directed toward refining the Django backend and React frontend to foster a scalable architecture and enrich the user experience. Moreover, integrating advanced machine learning models and refining user feedback mechanisms would aim to boost the precision and customization of job recommendations. By tackling these areas, the project team would strive to navigate past current hurdles, trim operational costs, and broaden the system's reach, securing its enduring value and relevance in the job market landscape.

References

[1]“Regular Expression (Regex) Tutorial,” *www3.ntu.edu.sg*. https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html#:~:text=A%20Regular%20Expression%20(or%20Regex

[2]O. Prunckun, “How Generative Text Artificial Intelligence (Specifically GPT) Works Under the Hood, In Plain…,” *Medium*, Oct. 25, 2023. https://orren.medium.com/how-generative-text-artificial-intelligence-specifically-gpt-works-under-the-hood-in-plain-0656bf19f1ce (accessed Apr. 07, 2024).

[3]O. Savage, “What is WebDriver in Selenium?,” *www.leapwork.com*. https://www.leapwork.com/blog/what-is-webdriver-in-selenium#:~:text=what%20it%20does.-

[4] F. Karabiber, “TF-idf - term frequency-inverse document frequency,” *Learn Data Science - Tutorials, Books, Courses, and More*, https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency/

[5]Z. Jaadi, “A step-by-step explanation of principal component analysis (PCA),” *Built In*, https://builtin.com/data-science/step-step-explanation-principal-component-analysis

[6][5] P. Khare, “Navigating clusters: Unveiling the optimal K in K-means clustering,” *Medium*, https://ai.plainenglish.io/navigating-clusters-unveiling-the-optimal-k-in-k-means-clustering-6f8ee8a1ec60

[7]A. Pant, “Introduction to logistic regression,” *Medium*, https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148

[8]F. Karabiber, “Cosine similarity,” *Learn Data Science - Tutorials, Books, Courses, and More*, https://www.learndatasci.com/glossary/cosine-similarity/

[9]Django, “The Web framework for perfectionists with deadlines | Django,” *Djangoproject.com*, 2019. https://www.djangoproject.com/

[10]“Free Databases - AWS,” *Amazon Web Services, Inc.* https://aws.amazon.com/free/database/?trk=b54801f4-057b-4340-b0a5-1ee26130ff8f&sc_channel=ps&ef_id=Cj0KCQjwiMmwBhDmARIsABeQ7xRAD2dnFgyBFYSxFe_HbT8pOM4Wz5dVmfdcAc81pAvGjf1pFWGy5c0aApmYEALw_wcB:G:s&s_kwcid=AL (accessed Apr. 08, 2024).

[11]“DB instance classes - Amazon Relational Database Service,” *docs.aws.amazon.com*. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.DBInstanceClass.html#Concepts.DBInstanceClass.Types

[12]“AWS Pricing Calculator,” *calculator.aws*. https://calculator.aws/#/createCalculator/RDSMySQL

[13]“Pricing,” openai.com. https://openai.com/pricing#language-models

[14]“Upgrading Git Large File Storage,” *GitHub Docs*. https://docs.github.com/en/billing/managing-billing-for-git-large-file-storage/upgrading-git-large-file-storage (accessed Apr. 08, 2024).